Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Certification of Compiled Assembly Code by Invariant Translation

Xavier Rival

École Normale Supérieure
45, rue d'Ulm
75 230, Paris, France
e-mail: rival@di.ens.fr

Abstract. We present a method for analyzing assembly programs obtained by compilation and checking safety properties on compiled programs. It proceeds by analyzing the source program, translating the invariant obtained at the source level and then by checking the soundness of the translated invariant with respect to the assembly program. This process is especially adapted to the certification of assembly or other machine-level kinds of programs. Furthermore, the success of the invariant checking enhance the level of confidence in the results of both the compilation and the static analysis. From the practical point of view, our method is generic in the choice of an abstract domain for representing sets of stores and the process does not interact with the compilation itself. Hence, a certification tool can be interfaced with an existing analyzer and designed so as to work with a class of compilers which do not need to be modified. Last, a prototype was implemented in order to validate the approach.

Keywords: Static program analysis; Certified compilation; Abstract Interpretation.

1 Introduction

Critical software is concerned with safety; hence, various static analysis methods have been developed and are applied to critical programs. However, these methods are usually applied to the source program and the source analysis may not be considered a trustable proof given the compiler may be incorrect and the compiled program may not be safe even if the source analysis succeeds in proving safety. Indeed modern compilers turn out to be very complex due to the size of their source code and to their perpetual evolution (for instance, the code of the current versions of **gcc** amounts to about 500 000 lines). Therefore, most critical applications like avionics require the certification of the form of the program which is actually executed, i.e. the assembly code itself.

Moreover, the safety properties of interest usually concern the very execution of the program; hence, checking it on the compiled program (i.e. the version that is actually executed) yields more trustable proofs of safety. For instance, the semantics of errors is defined at the machine level first. The memory access errors (out-of-bound array index or void pointer dereference in C programs) are the source language counterpart for some assembly errors (attempt to access a wrong part of memory). If we prove that a source C program does not yield any memory access error, then we can deduce that a compiled form of this program is memory safe only under some additional assumptions, i.e. mainly that the program is compiled in a correct way for some definition of "correct" which should be made explicit and that the memory allocation is done at the assembly level in a safe way, which should also be made explicit. Furthermore, the nature of the undesirable behaviors may be compiler or even architecture dependent, as is the case for overflows: The size of registers depends on the target processor and the way integer data types are compiled affects the overflows that occur in the compiled program (this is especially true for data types that do not correspond to the size of registers like short integer data types). Languages like C leave many error cases as unspecified in order to let the compiler implementator free when designing more optimizations: For example, an out-of-bound array index in a C program results in an undefined behavior, which may be an immediate error or a wrong, yet continued execution. Therefore, checking safety properties at the assembly level is noticeably advantageous – in particular when dealing with highly critical software.

As a way to achieve that, we may envisage certifying the assembly program directly. However, analyzing directly and efficiently precise high-level properties of

assembly programs may be quite difficult due to a loss of structure at compile time. In particular, the control structure of assembly programs is based on gotos, which are much more complicated to analyze than loops. Static analysis methods for improving speed and precision apply in an easier way to well-structured loops than general control flow graphs. Furthermore, the data structures (like arrays, records or enums) are translated into more complicated assembly structures since everything turns into a sequence of memory cells and low-level details should be taken into account (as memory cells alignments). In the other hand, the formal (semi-automatic) proof of a full C compiler cannot be envisaged on account of the work task that would be involved in such a project and because any modification or evolution of the compiler would make the proof out-of-date (proving a commercial compiler is not a realistic solution). The last limitation also applies to a system that would translate a proof of safety at compile time.

The solution proposed here is to analyze the source version of the program using an automatic tool and to derive automatically a "candidate invariant" for the assembly program. This invariant is obtained by translating the source invariant thanks to some information about the way the program is compiled (in most cases, this additional information can be found in the debugging information provided by the compiler, which describes the correspondence between source and target variables and program points). Then, an automatic tool checks that the candidate invariant is semantically sound: it is an upper-approximation of the set of reachable states of the program. If the program P_c is obtained by compiling the program P_s the method proceeds as follows: A source analyzer generates an invariant $\mathcal{P}_s$ for the source program and an external tool derives the candidate invariant $\mathcal{P}_c$; then an assembly checker attempts to prove that the property $\mathcal{P}_c$ holds for the program P_c. Afterwards, the property $\mathcal{P}_c$ can be used for verifying that P_c satisfies the desired safety properties. It can be noticed that this approach allows to take benefit from existing fast and precise source analyzers (like those of [BCC+02, BCC+03]). Our method does not require the instrumentation of the compiler; in case the debugging information format is standard, we can even consider designing a tool that would translate invariants for certifying assembly programs produced by a class of compilers. Moreover, we need to cope with the specificities of assembly programs for the checking of invariants only and not for their inference. When the checking succeeds, the translated invariant can be considered correct under only one assumption: The checker must be correct. Therefore the security level achieved by this approach is the same as those of a direct analysis of the assembly code. Moreover the success of the checking entails a correctness result about the compilation: The target program presents similar behaviors as the source program (in the abstract semantics point of view). In the other hand the method is incomplete: A failure of the invariant checking does not entail that the compiler is buggy; it may be due to a loss of precision at the translation time or at the checking time. The approach proposed here is formalized inside the Abstract Interpretation frameworks [CC77, CC79], which provides an integrated view in a single framework of both static analysis [Cou81, BCC+03] and program transformations [CC02] (hence, compilation). Furthermore, we validated our approach by designing a prototype aimed at checking the absence of runtime errors and undefined behaviors in PowerPC assembly programs obtained by compiling realistic C programs. Our choice of the C language was justified by the use of this language in safety critical systems.

Plan. Section 2 presents preliminaries and describes the source and the assembly languages which are considered in the following of the paper. We formalize the compilation correctness in Section 3. Section 4 describes a class of static analyses practically large enough for answering most of the safety questions about imperative source programs and shows how an invariant can be derived at the assembly level from a source invariant. Section 5 discusses the problem of checking the translated invariant independently of the source analysis. We detail the practical problems that arise when checking the invariant at the assembly level in Section 6. The prototype we implemented is described in Section 7. Section 8 concludes.

Related work. Most attempts to proving formally a compiler concentrated on rather high-level languages and byte code assembly languages [Str02] or to toy compilers written for that purpose [Ber98]. The lack of automation of theorem provers severely limits the possibility of proving large programs in general and compilers in particular.

Among direct static analyses of assembly programs, we can cite the determination of properties about the cache usage (cache misses and cache hits) presented in [AFMW96], the analysis of pipeline behavior of [TF98] and the combination of these two analyses in [TFW00]: Precise information could be inferred about the worst case execution time of assembly programs by taking into account many complex aspects of the architecture. However, we are not aware of any example of direct analysis for high-level properties at the assembly level.

The idea to translate at compile time semantic information about the source program into information about the assembly program was developed in the Proof-Carrying Code approach described in [Nec97, App01]. In this approach an untrusted compiler is supposed to provide annotations with the assembly code it produces. Before it executes the target program, the code consumer generates Verification Conditions so as to check that the assembly program does not violate the safety policy and attempts to prove it using the annotations supplied by the compiler. If it succeeds then the assem-

bly code obeys the safety policy and can be executed safely. The compiler of [NL98] implements this methodology: In this case, the compiler annotations are type information.

A Typed Intermediate Language (TIL) was proposed in [MTC+96, TMC+96] as a means to keep information about source ML programs in order to make further optimizations possible and trustable. Basically, well-typed programs should not produce certain types of errors (the memory allocation should be safe). This methodology was extended to a Typed Assembly Language (TAL) in [MCG+99]: The purpose of this work was also to design a safe compiler for a type-safe subset of C. However changing to a safe subset of the C language is not always possible in the case of embedded systems. Furthermore, enforcing safety through typing systems may turn out somewhat difficult in some cases: In particular, handling overflows is not very natural in the context of typing systems. Last, the implementation of a specific certifying compiler involves a sizeable task.

Another approach to certified compilation proceeds by proving the correctness of each compilation separately. When a program P_s is compiled into a program P_c, an external tool generates proof obligations so as to prove that P_c is equivalent to P_s for some definition of "is equivalent to". This method known as *Translation Validation* was pioneered by [PSS98], and then implemented in [Nec00] and extended in [ZPFG02]. Translation Validation provides proofs of compilation correctness for a rather concrete semantic interpretation of programs. However the goal of this approach is not to produce safety proofs for assembly programs.

Our work on Invariant Translation was developed in a previous contribution [Riv03]. The purpose is to translate abstract invariants computed at the source level, using static analyzers which are similar to those presented in [BCC+02, BCC+03] in order to derive proofs of safety for compiled programs in the context of critical embedded systems. The Invariant Translation also yields a kind of abstract proof for the compilation: In case it succeeds, it proves that compilation preserves some abstract property of the source program. Yet, it is less adapted to proving a strong operational equivalence between source and target programs than Translation Validation: The latter operates at a rather concrete semantic level; hence, it aims at proving a stronger equivalence.

2 Preliminaries and Notations

This section presents some basic notations we use in the following; it also introduces the syntax and semantics of the typical source and assembly languages which we consider along the paper.

2.1 Mathematical Common Notations

We write $\mathbb{Z}$ for the set of positive and negative integers ($\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$) and $\mathbb{B}$ for the set of booleans: $\mathbb{B} = \{\mathcal{T}, \mathcal{F}\}$, where $\mathcal{T}$ and $\mathcal{F}$ respectively denote true and false.

When necessary, we write Ω for erroneous behaviors. If $\mathcal{E}$ is a set, we write $\mathcal{E}_\Omega$ for the set $\mathcal{E} \cup \{\Omega\}$ (for instance, $\mathbb{Z}_\Omega$, $\mathbb{B}_\Omega$).

When taking overflows into account, we will write $\mathbb{Z}^o$ for the set of machine representable integers $\{n \in \mathbb{Z} \mid N_{\min} \leq n \leq N_{\max}\}$ where $N_{\min}$ and $N_{\max}$ are the smallest and the biggest representable integers.

In the following, if $\mathcal{E}$ is a set, we will write $\mathbb{P}(\mathcal{E})$ for the set of the subsets of $\mathcal{E}$ ($\mathbb{P}(\mathcal{E}) = \{X \mid X \subseteq \mathcal{E}\}$). If $x_0, \ldots, x_n$ are elements of $\mathcal{E}$, then we write $\langle x_0, \ldots, x_n \rangle$ for the finite sequence composed by these elements (a sequence is a function from an interval of integers starting from 0 like $\{0, 1, \ldots, n\}$ to a set $\mathcal{E}$). The set of finite sequences of elements of $\mathcal{E}$ is denoted by $\mathcal{E}^\star$.

If $\mathcal{E}$ and $\mathcal{F}$ are sets, then we write $\mathcal{E} \to \mathcal{F}$ for the set of functions from $\mathcal{E}$ to $\mathcal{F}$. If $f \in \mathcal{E} \to \mathcal{F}$, then we let $\widehat{f}$ denote the function defined by $\widehat{f} : \mathbb{P}(\mathcal{E}) \to \mathbb{P}(\mathcal{F})$; $X \mapsto \{f(x) \mid x \in X\}$. Furthermore, if $\sqsubseteq$ is an order relation over $\mathcal{F}$, then the pointwise extension of $\sqsubseteq$ to $\mathcal{E} \to \mathcal{F}$ is denoted by $\dot{\sqsubseteq}$. We recall that a *lattice* is an ordering $(\mathcal{E}, \sqsubseteq)$ with a minimal and a maximal element and with a binary lower upper bound operator and a binary greater lower bound operator. In case any subset of $\mathcal{E}$ has a greater lower bound and a lower upper bound, we say that $(\mathcal{E}, \sqsubseteq)$ is a *complete lattice*.

We sometimes use the lambda notation to denote functions: $\lambda x \in \mathcal{E}.e$ simply stands for the function $\mathcal{E} \to \mathcal{F},\ x \mapsto e$.

2.2 Abstract Interpretation and Program Transformations

Abstract Interpretation [CC77, CC79] was developed as a way of deriving relationships between different semantics so as to provide approximate but computable answers to undecidable or costly problems. The approximations preserve logical soundness. An abstract semantics is often less expressive than the standard semantics; hence, considering abstract properties may induce a loss of precision (some properties cannot be deduced or stated any more), but reasoning in the abstract is still sound (furthermore, it should be computer tractable).

In practice, the *concrete semantics* $[\![P]\!]$ provides the most precise description of the behavior of P. It is generally defined as an element of a complete lattice $(D, \sqsubseteq)$. An abstract domain is simply another complete lattice $(D^\sharp, \sqsubseteq)$. A *Galois connection* between the concrete domain D and the abstract domain $D^\sharp$ is a pair of functions $\alpha : D \to D^\sharp$, $\gamma : D^\sharp \to D$ such that $\forall x \in D, y \in D^\sharp,\ \alpha(x) \sqsubseteq y \Leftrightarrow x \sqsubseteq \gamma(y)$. The intuitive

meaning of $x \sqsubseteq \gamma(y)$ is that y is a sound abstract approximation of the concrete property x; i.e. the concrete property x entails the abstract property y. The abstract semantics $\alpha([\![P]\!])$ of the program P will be denoted by $[\![P]\!]^\sharp$.

If (α, γ) is a Galois connection, we will write $D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\sharp$. In some cases, this formalization of abstraction does not apply. In particular, the existence of an abstraction function α may not be achieved in case some concrete element does not enjoy a "best abstract property". For instance a disk does not have a best abstract approximation in the domain of polyhedra [CH78]: The abstraction relation between the domain $\mathbb{P}(\mathbb{R}^2)$ and the domain of convex polyhedra features no abstraction function α (only a concretization γ can be defined). Other (more general) ways of formalizing the notion of abstraction can be found in [CC92]; however, we consider in this paper *Galois connection-based* abstract interpretations only for the sake of simplicity.

The semantics $[\![P]\!]$ can in general be defined as the least fixpoint of a monotone semantic function F in the lattice D. 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 complete lattice $D^\sharp$ as shown by the fixpoint transfer theorem of [Tar55]. However, in most cases, the abstract semantics itself is not computable either because the iteration is infinite or because $F^\sharp$ is not computable or just because there is no function satisfying the above equality. Then a sound approximation of $[\![P]\!]^\sharp$ is derived by computing the least fixpoint of a computable function $F^\sharp$ such that $\alpha \circ F \sqsubseteq F^\sharp \circ \alpha$ or by using a widening operator or by applying both techniques.

Furthermore, Abstract Interpretation proved useful in studying program transformations [CC02]. Formalizing a program transformation t defined syntactically proceeds by defining suitable semantic observations $[\![.]\!]^o_s$ and $[\![.]\!]^o_t$ of the standard semantics for source $[\![.]\!]_s$ and target programs $[\![.]\!]_t$, which should express the properties preserved by the transformation t. In most cases, these observational semantics can be defined as abstractions of the standard semantics: $[\![P_i]\!]^o_i = \alpha_i([\![P_i]\!]_i)$ for $i \in \{s, t\}$. Hence the correctness of the transformation boils down to $P_t = t(P_s) \Longrightarrow \alpha_t([\![P_t]\!]_t) \equiv \alpha_s([\![P_s]\!]_s)$ where $\equiv$ corresponds to some kind of bijection. In this context, relating semantics in hierarchies of abstract interpretations [Cou97] is particularly useful.

We formalize both static analysis and compilation in the Abstract Interpretation framework first, and then we state our methodology in this framework.

2.3 Programs, Semantics

In this paper, we consider imperative programming languages only. An *execution state* is a pair (l, ρ) where l is a program point (or *label*) and ρ is a store. A program is defined by the data of a set of labels, a set of stores and a transition relation which specifies the way one steps from a state to another state:

Definition 1 (Transition system associated to a program). Let R be a set of values for variables. The *transition system* associated to a *program* P is a tuple $(L_P, V_P, i_P, \rightarrow_P)$ where:

- L_P is the set of *labels* of P;
- V_P is the set of *memory locations* of P; the corresponding set of stores $V_P \rightarrow R$ is denoted by S_P; the set of states for program P is denoted by $E_P = L_P \times S_P$
- i_P is the *entry program point* of P: It is the label any execution of P starts at;
- $(\rightarrow_P) \subseteq E_P \times E_P$ is the *transition relation* of P. Intuitively, $(l, \rho) \rightarrow_P (l', \rho')$ means that if an execution of P reaches point l with store ρ, then it may continue at point l' with store ρ'.

Note that a program point is not necessarily a syntactic program point: In case of procedural programs, a label l would define a pair (κ, l_s) where κ is a stack and l_s is a syntactic program point.

In general, we add an error state denoted by Ω to the set of states of transition systems: $E_P = \{\Omega\} \cup L_P \times S_P$. No transition starts from Ω given this state is blocking, so $(\rightarrow_P) \subseteq (E_P \setminus \{\Omega\}) \times E_P$.

An execution trace of a program is a finite sequence of states, starting at the entry program point and such that one steps from a state to the next one according to the transition relation. The trace corresponding to the sequence of states $e_0, \ldots, e_n$ is noted $\langle e_0, \ldots, e_n \rangle$. One can remark that our presentation allows non-determinism since $\rightarrow_P$ is a relation (in the deterministic case, it would turn into a function). The semantics of a program P is the set of the execution traces of P. It is formally defined as follows:

Definition 2 (Semantics of a program). The *concrete semantic function* of the program P is the function F_P defined by:

$$\begin{array}{rcl} F_P : \mathbb{P}(E_P^\star) & \longrightarrow & \mathbb{P}(E_P^\star) \\ X & \longmapsto & \{\langle (l_0, \rho_0), \ldots, (l_n, \rho_n), (l_{n+1}, \rho_{n+1}) \rangle \\ & & \quad \mid \langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle \in X \\ & & \quad \wedge (l_n, \rho_n) \rightarrow_P (l_{n+1}, \rho_{n+1}) \} \\ & & \cup \{\langle (i_P, \rho) \rangle \mid \rho \in S_P\} \end{array}$$

Then, the *semantics* of the program P is the least-fixpoint of F_P:

$$[\![P]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset} F_P$$

Note that the operator F_P is continuous, hence the least-fixpoint exists and can be written as follows:

$$\mathrm{lfp}^{\subseteq}_{\emptyset} F_P = \bigcup_{n \in \mathbb{N}} F_P^n(\emptyset)$$

(in other words the computation of the fixpoint does not require a transfinite iteration)

$$
\begin{array}{lll}
\texttt{Lv} & ::= & x \ \ (x \in X) \mid x[\texttt{E}] \ \ (x \in X) \\
\texttt{E} & ::= & n \ \ (n \in \mathbb{Z}^o) \mid \texttt{Lv} \\
 & \mid & \texttt{E} + \texttt{E} \mid \texttt{E} - \texttt{E} \mid \texttt{E} \star \texttt{E} \mid \texttt{E}/\texttt{E} \\
\texttt{C} & ::= & \textbf{true} \mid \textbf{false} \mid \neg \texttt{C} \mid \texttt{C} \wedge \texttt{C} \\
 & \mid & \texttt{C} \vee \texttt{C} \mid \texttt{E} == \texttt{E} \mid \texttt{E} < \texttt{E} \\
\texttt{S} & ::= & \texttt{Lv} := \texttt{E} \mid \textbf{if}(\,\texttt{C}\,)\ \texttt{B}\ \textbf{else}\ \texttt{B} \mid \textbf{while}(\,\texttt{C}\,)\ \texttt{B} \\
\texttt{B} & ::= & \{\,\texttt{S}; \ldots ; \texttt{S}\}
\end{array}
$$

Fig. 1. A simple imperative language

2.4 A Simple Imperative Language

We present here the source language we consider in the following. The grammar is given on figure 1. It features integer variables and arrays, simple assignments, conditionals and loops. A memory location $v \in V$ is either a variable or an array cell. We write X for the set of objects (arrays and integer variables). Expressions have integer type; hence, we consider that $R = \mathbb{Z}^o$.

The semantics of expressions and conditions are defined as follows:

$$
\begin{array}{c}
\forall e \in \texttt{E},\ [\![e]\!] \in (V \to \mathbb{Z}^o) \to \mathbb{Z}^o_\Omega \\
\forall c \in \texttt{C},\ [\![c]\!] \in \mathbb{B}_\Omega \to \mathbb{P}(V \to \mathbb{Z}^o)
\end{array}
$$

Intuitively, the semantics of an expression maps a store to a value or to the error constant Ω in case an error happens when the expression is evaluated. It relies on a definition of an interpretation $\check{\oplus} : \mathbb{Z}^o_\Omega \times \mathbb{Z}^o_\Omega \to \mathbb{Z}^o_\Omega$ for the operator $\oplus \in \{+, -, \star, /\}$. The interpretation $\check{\oplus}$ of the operator $\oplus$ is assumed to be Ω-strict: $\forall v \in \mathbb{Z}_\Omega,\ v \check{\oplus} \Omega = \Omega \check{\oplus} v = \Omega$ (intuitively, an error is always propagated). The interpretations of the binary operators are supposed to handle error cases as division by 0 and overflows: For instance, $\check{/}(v, 0) = \check{+}(N_{\max}, 1) = \Omega$. The semantics of expressions is defined by induction on the syntax as follows (ρ denotes an environment, x a variable, t an array of length n; e_0, e_1 denote expressions):

$$
\begin{array}{rl}
[\![n]\!](\rho) & = n \\
[\![x]\!](\rho) & = \rho(x) \\
[\![t[e_0]]\!](\rho) & = \begin{cases} \rho(t\,[[\![e_0]\!](\rho)]) & \text{if } 0 \leq [\![e_0]\!](\rho) < n \\ \Omega & \text{otherwise} \end{cases} \\
[\![e_0 \oplus e_1]\!](\rho) & = \check{\oplus}([\![e_0]\!](\rho), [\![e_1]\!](\rho)) \quad \oplus \in \{+, -, \star, /\}
\end{array}
$$

Accommodating non-determinism would require considering sets of values instead of values here.

The semantics of a condition c maps a value $b \in \mathbb{B}_\Omega$ to the set of stores in which the condition c evaluates to b. The usual interpretations of the logical and comparison operators are lifted to Ω-strict interpretations. As for the expressions, the semantics of conditions is defined by induction on the syntax. We give only a few cases (c_0 and c_1 denote conditional expressions):

$$
\begin{array}{rl}
[\![\textbf{true}]\!](\mathcal{T}) & = V \to \mathbb{Z} \\
[\![\textbf{true}]\!](\mathcal{F}) & = [\![\textbf{true}]\!](\Omega) = \emptyset \\
[\![c_0 \wedge c_1]\!](\mathcal{T}) & = [\![c_0]\!](\mathcal{T}) \cap [\![c_1]\!](\mathcal{T}) \\
[\![c_0 \wedge c_1]\!](\mathcal{F}) & = [\![c_0]\!](\mathcal{F}) \cap [\![c_1]\!](\mathcal{F}) \cup [\![c_0]\!](\mathcal{T}) \cap [\![c_1]\!](\mathcal{F}) \\
 & \quad \cup [\![c_0]\!](\mathcal{F}) \cap [\![c_1]\!](\mathcal{T}) \\
[\![c_0 \wedge c_1]\!](\Omega) & = [\![c_0]\!](\Omega) \cup [\![c_1]\!](\Omega)
\end{array}
$$

We suppose that each statement s is associated to a label l (which intuitively denotes the program point right before the statement s). The transition system of a program P is defined by the set of labels associated to the statements of P and by the transition relation defined below by considering all the statements in P:

- Case of an assignment $l : t\,[e_0] := e_1; l' : \ldots$ (where t is an array of size n):
  . if $[\![e_0]\!](\rho) \neq \Omega$ and $[\![e_1]\!](\rho) \neq \Omega$ and $0 \leq [\![e_0]\!](\rho) < n$, then:
  $$(l, \rho) \to_P (l', \rho[t\,[[\![e_0]\!](\rho)] \leftarrow [\![e_1]\!](\rho)])$$
  . else,
  $$(l, \rho) \to_P \Omega$$
  An affectation to a variable is similar.
- Case of a conditional $l :\ \textbf{if}(c)\,\{l_t : B_t;\ l'_t\}\ \textbf{else}\,\{l_f : B_f;\ l'_f\};\ l' : \ldots$:
$$
\begin{array}{c}
\rho \in [\![c]\!](\mathcal{T}) \Longrightarrow (l, \rho) \to_P (l_t, \rho) \\
\rho \in [\![c]\!](\mathcal{F}) \Longrightarrow (l, \rho) \to_P (l_f, \rho) \\
\rho \in [\![c]\!](\Omega) \Longrightarrow (l, \rho) \to_P \Omega \\
(l'_i, \rho) \to_P (l', \rho) \quad \text{where } l'_i \in \{l'_t, l'_f\}
\end{array}
$$
- Case of a loop $l :\ \textbf{while}(c)\,\{l_b : B_b; l'_b\};\ l' : \ldots$:
$$
\begin{array}{c}
\rho \in [\![c]\!](\mathcal{T}) \Longrightarrow (l, \rho) \to_P (l_b, \rho) \\
\rho \in [\![c]\!](\mathcal{F}) \Longrightarrow (l, \rho) \to_P (l', \rho) \\
\rho \in [\![c]\!](\Omega) \Longrightarrow (l, \rho) \to_P \Omega \\
(l'_b, \rho) \to_P (l, \rho)
\end{array}
$$

This language could be extended to a procedural subset of the C language very easily. The definition of the semantics would be similar (labels would include a calling context as mentioned in section 2.3). The extension to non-determinism would also be trivial.

2.5 A Simple Assembly Language

This subsection describes the simple (yet realistic) assembly language, we consider in this paper. It corresponds to a (very) simplified model of the assembly language of the PowerPC processor [Mot97] (the prototype presented in section 7 was designed for the real PowerPC execution model).

The simplified execution model features a given number of integer registers denoted by $\mathrm{r}_0, \ldots, \mathrm{r}_N$, and access to memory with integer addresses. An assembly program is a sequence of labeled instructions (we simply define the label of an instruction as the value of the program counter before this instruction is executed). The syntax of instructions is given on figure 2.

As in many processors, a conditional branching is decomposed in several steps: The comparison instruction sets the value of a so-called "condition register" cr (possible values for cr are LT, EQ and GT: LT means "less than"; EQ means "equal"; GT means "greater than");

$$
\begin{array}{lcl}
n & \in & \mathbb{Z}^o \\
c & \in & \{<, \leq, =, \neq, >, \geq\} \\
v & \in & \{\mathrm{r}_0, \ldots, \mathrm{r}_N\} \cup \mathbb{Z}^o \\
\texttt{op} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \\
\texttt{I} & ::= & \texttt{load}\ \mathrm{r}_0,\ n\,(v) \\
 & \mid & \texttt{store}\ \mathrm{r}_0,\ n\,(v) \\
 & \mid & \texttt{li}\ \mathrm{r}_0,\ n \\
 & \mid & \texttt{op}\ \mathrm{r}_0,\ \mathrm{r}_1,\ \mathrm{r}_2 \mid \texttt{mr}\ \mathrm{r}_0,\ \mathrm{r}_1 \\
 & \mid & \texttt{cmp}\ \mathrm{r}_0,\ \mathrm{r}_1 \\
 & \mid & \texttt{bc}(c)\ l \mid \texttt{b}\ l
\end{array}
$$

Fig. 2. A simple assembly language

the conditional branching instruction directs the execution according to the condition register value. We write $\mathbb{C}$ for the set $\{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$. Hence, we consider here the set of values $R = \mathbb{Z}^o \cup \mathbb{C}$.

The address of a variable x stored in the memory is denoted by $\underline{x}$. We write $M\{n\}$ for the memory cell of address n, where $n \in \mathbb{N}$. As is the case for many real architectures, memory access proceeds by relative addressing: The instruction $\texttt{load}\ \mathrm{r}_0,\ n\,(v)$ loads the content of the memory cell of address $n + v$ into the register r_0.

The transition system associated to a program P is defined by the labels of all the instructions of the program and the transition relation defined below by considering all the instructions in the program ($l, l', l'', \ldots$ denote program points):

- the "load integer" instruction $l : \texttt{li}\ \mathrm{r}_0,\ n;\ l' : \ldots$ loads the integer n into the register r_0:

$$(l, \rho) \rightarrow_P (l', \rho[\mathrm{r}_0 \leftarrow n])$$

- the "load" instruction $l : \texttt{load}\ \mathrm{r}_0,\ \underline{x}\,(v);\ l' : \ldots$ loads the content of the memory cell of address $\underline{x} + v$ (v is either an integer constant or the content of a register) if $\underline{x} + v$ is a valid address (if not, it fails):
  . If $\underline{x} + v$ is a valid address, then:

$$(l, \rho) \rightarrow_P (l', \rho[\mathrm{r}_0 \leftarrow \rho(M\{\underline{x} + v\})])$$

  . If $\underline{x} + v$ is not a valid address, then:

$$(l, \rho) \rightarrow_P \Omega$$

- the "store" instruction $l : \texttt{store}\ \mathrm{r}_0,\ \underline{x}\,(v);\ l' : \ldots$ stores the content of the register r_0 into the memory cell of address $\underline{x} + v$ if $\underline{x} + v$ is a valid address (if not, it fails):
  . If $\underline{x} + v$ is a valid address, then:

$$(l, \rho) \rightarrow_P (l', \rho[M\{\underline{x} + v\} \leftarrow \rho(\mathrm{r}_0)])$$

  . If $\underline{x} + v$ is not a valid address, then:

$$(l, \rho) \rightarrow_P \Omega$$

- the "move register" instruction $l : \texttt{mr}\ \mathrm{r}_0,\ \mathrm{r}_1 : l' : \ldots$ copies the content of the register r_0 into the register r_1:

$$(l, \rho) \rightarrow_P (l', \rho[\mathrm{r}_0 \leftarrow \rho(\mathrm{r}_1)])$$

- the "compare" instruction $l : \texttt{cmp}\ \mathrm{r}_0,\ \mathrm{r}_1;\ l' : \ldots$ compares the content v_0 of the register r_0 with the content v_1 of the register r_1; if $v_0 < v_1$, then the value of the condition register is set to LT; if $v_0 = v_1$, then the value of the condition register is set to EQ; if $v_0 > v_1$, then the value of the condition register is set to GT:

$$
\begin{array}{ll}
\text{if } \rho(\mathrm{r}_0) < \rho(\mathrm{r}_1) \text{ then}, & (l, \rho) \rightarrow_P (l', \rho[\mathrm{cr} \leftarrow \mathrm{LT}]) \\
\text{if } \rho(\mathrm{r}_0) = \rho(\mathrm{r}_1) \text{ then}, & (l, \rho) \rightarrow_P (l', \rho[\mathrm{cr} \leftarrow \mathrm{EQ}]) \\
\text{if } \rho(\mathrm{r}_0) > \rho(\mathrm{r}_1) \text{ then}, & (l, \rho) \rightarrow_P (l', \rho[\mathrm{cr} \leftarrow \mathrm{GT}])
\end{array}
$$

- the "conditional branching" instruction $l : \texttt{bc}(<)\ l'';\ l' : \ldots$ branches to l'' or to the next instruction depending on the value stored in the condition register (the case of $\texttt{bc}(c)\ l''$ where c is any condition is similar):

$$
\begin{array}{ll}
\text{if } \rho(\mathrm{cr}) = \mathrm{LT} \text{ then}, & (l, \rho) \rightarrow_P (l'', \rho) \\
\text{if } \rho(\mathrm{cr}) \in \{\mathrm{EQ}, \mathrm{GT}\} \text{ then}, & (l, \rho) \rightarrow_P (l', \rho)
\end{array}
$$

- the "branching" instruction $l : \texttt{b}\ l'';\ l' : \ldots$ branches to label l'':

$$(l, \rho) \rightarrow_P (l'', \rho)$$

- the "addition" instruction $l : \texttt{add}\ \mathrm{r}_0,\ \mathrm{r}_1,\ \mathrm{r}_2;\ l' : \ldots$ adds the content of the registers r_1 and r_2; then, if no error occurs, it stores the result into the register r_0; in case an error occurs (i.e. the result is Ω) the operation instruction evaluates to the error state:
  . If $\check{+}(\rho(\mathrm{r}_1), \rho(\mathrm{r}_2)) \neq \Omega$, then:

$$(l, \rho) \rightarrow_P (l', \rho[\mathrm{r}_0 \leftarrow \check{+}(\rho(\mathrm{r}_1), \rho(\mathrm{r}_2))])$$

  . If $\check{+}(\rho(\mathrm{r}_1), \rho(\mathrm{r}_2)) = \Omega$, then:

$$(l, \rho) \rightarrow_P \Omega$$

  The case of the other arithmetic instructions `mul`, `sub`, `div` is similar. Note that the interpretations of the arithmetic operators are the same as for the source language of section 2.4.

This simple assembly language could be extended to handle procedures. Then, we would have to extend the assembly model by taking into account the execution stack. Some extra instructions would allow to update the stack at the function calls and returns.

3 Compilation as a Program Transformation

This section attempts to formalize the compilation of a source program P_s into an assembly program P_a. This is achieved by defining a suitable observational semantics for source and target programs, which states an equivalence between them.

3.1 Intuition about Compilation

The purpose here is to state how we expect source and compiled programs to be related: Indeed, proving properties about compiled programs from properties of source programs requires a notion of "correct compilation". Intuitively, both programs should carry out the same computations, that is the execution traces of both programs should be isomorphic. We consider here the case of imperative source programming languages.

We assume that the program P_s is compiled into the program P_c. If the compilation is correct and if the execution of a statement in P_s starting at a state (l_s, ρ_s) ends in a state (l'_s, ρ'_s), then there should exist two states (l_c, ρ_c) and (l'_c, ρ'_c) in the compiled program that are respectively "related" to (l_s, ρ_s) and (l'_s, ρ'_s) and such that $(l_c, \rho_c) \rightarrow (l'_c, \rho'_c)$ in one or several assembly execution steps. Similarly, any sequence of execution of the compiled program should have a counterpart in the source program. Describing the link between the executions of both programs is the purpose of this section.

The relation between program points of "related states" states some kind of equivalence between the control structures of both programs. The relation between stores of "related states" asserts that some source and assembly memory locations are in correspondence; hence, they should store the same value –modulo some convention about the machine representation of the source values.

For instance, in the case of the example given on figure 3, the assembly counterpart for the source variable x is the memory cell of address $\underline{x}$, whereas the registers have no source counterpart. The assembly program point l^a_2 corresponds to the source program point l^s_1 (and the same for the other pairs of program points listed on figure 3(c)), whereas some assembly program points have no source counterpart: For example, the label l^a_1 cannot be mapped to any point in the source program. At the semantics level, the compilation of P_s into P_a is correct for this mapping of the source and assembly program points and memory locations. The correctness of compilation expresses for instance that, if x has value v at point l^s_1 for some execution σ_s of P_s, then there exists some execution σ_a of P_a which reaches point l^a_2 and such that the value contained in $\underline{x}$ at this point is equal to v. Furthermore, σ_s and σ_a present the same transitions: If σ_s steps forward from the state $(l^s_1, [x \mapsto v])$ to the program point l^s_2 (i.e. σ_s enters the loop), then σ_a carries out a corresponding step (or sequence of steps) from $(l^a_2, [\ldots, x \mapsto v])$ to l^a_6; hence, it proceeds through the execution path $\langle l^a_2, l^a_3, l^a_4, l^a_5, l^a_6 \rangle$ (i.e. it does not follow the branching to l^a_{11}).

In general one source statement is compiled into a sequence of assembly statements; therefore some intermediate program points in the assembly program do not enjoy a counterpart in the source, as remarked above in the case of the example. Similarly some memory locations of the assembly program do not correspond to any memory location of the source program as is the case of the registers. Furthermore, some basic compiler optimizations may remove dead code or dead variables; hence, some source program points or memory locations may not have a counterpart in the compiled program.

Consequently, the relation between the source and the compiled program can only be formulated on a "restricted" form of the semantics, which ignores some parts of the computation. We detail in the following subsection the observational semantics we will use in order to define the correctness of compilation.

3.2 Observational Semantics

We consider here a program P defined by the labeled transition system $(L_P, V_P, i_P, \rightarrow_P)$ and by the set of variables V_P. The notions presented here will be instantiated to both source and assembly programs in the following.

Let $L^r_P \subseteq L_P$ and $V^r_P \subseteq V_P$ be "restricted" sets of program points and memory locations. The set L^r_P intuitively represents the program points we want to observe; similarly, V^r_P stands for the set of memory locations we want to keep. Furthermore, the notation S^r_P stands for $V^r_P \rightarrow R$; it denotes the set of the stores which assign a value to the memory locations in L^r_P. We first define projections for stores and for program points; then, the observational semantics will be defined as the projection of all the traces of $[\![P]\!]$.

Store restriction. The *store projection operator* ϕ maps a store $\rho \in S_P$ to a "restricted store" $\rho' \in S^r_P$:

$$\begin{array}{rl} \phi : S_P & \longrightarrow S^r_P \\ \rho & \longmapsto \rho' = \lambda x \in V^r_P . \rho(x) \end{array}$$

Trace restriction to a set of program points. The *trace projection operator* Φ forgets about the states (l, ρ) such that l does not belong to L^r_P, and applies the store projection operator to the stores of the remaining states. The states (l, ρ) such that l belongs to L^r_P are kept in the same order as they appear in the original sequence. More formally, Φ is defined as follows:

$$\Phi : (L_P \times S_P)^\star \longrightarrow (L^r_P \times S^r_P)^\star$$

If $\sigma = \langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle$, then $\Phi(\sigma) = \sigma'$ where:

$$\begin{cases} \sigma' = \langle (l_{k_0}, \phi(\rho_{k_0})), \ldots, (l_{k_m}, \phi(\rho_{k_m})) \rangle \\ 0 \leq k_0 < \ldots < k_m \leq n \\ \{k_0, \ldots, k_m\} = \{i \mid (0 \leq i \leq n) \wedge (l_i \in L^r_P)\} \end{cases}$$

We envisage here a trace of the example assembly program of figure 3(b):

Example 1. As the mapping presented in figure 3(c) suggests, we choose $L^r_a = \{l^a_0, l^a_2, l^a_6, l^a_{10}, l^a_{11}\}$ and $V^r_a = \{\underline{x}\}$.

$V_{P_s} = \{x\}$
$l_0^s : x := 0;$
$l_1^s : \textbf{while}(x < 100)\ \{$
$l_2^s : \quad x := x + 1;$
$l_3^s : \}$
$l_4^s : \langle \text{end of the program} \rangle$

(a) Source program P_s

$V_{P_a} = \{\underline{x}\}$

l_0^a : `li` r_0, 0	l_6^a : `load` r_0, $\underline{x}(0)$
l_1^a : `store` r_0, $\underline{x}(0)$	l_7^a : `li` r_1, 1
l_2^a : `load` r_0, $\underline{x}(0)$	l_8^a : `add` r_2, r_0, r_1
l_3^a : `li` r_1, 100	l_9^a : `store` r_2, $\underline{x}(0)$
l_4^a : `cmp` r_0, r_1	l_{10}^a : `b` l_2^a
l_5^a : `bc`$(\geq)$ l_{11}^a	l_{11}^a : $\langle \text{end of the program} \rangle$

(b) Assembly program P_a

. Program points:
$l_0^s \longleftrightarrow l_0^a$
$l_1^s \longleftrightarrow l_2^a$
$l_2^s \longleftrightarrow l_6^a$
$l_3^s \longleftrightarrow l_{10}^a$
$l_4^s \longleftrightarrow l_{11}^a$
. Memory locations: $x \longleftrightarrow \underline{x}$

(c) Relation

Fig. 3. An example of compilation

Let σ_a represent the very beginning of an execution of P_a:

$$\langle\ (l_0^a, [\underline{x} \mapsto v, r_0 \mapsto v_0, r_1 \mapsto v_1, \ldots]),\\ (l_1^a, [\underline{x} \mapsto v, r_0 \mapsto 0, r_1 \mapsto v_1, \ldots]),\\ (l_2^a, [\underline{x} \mapsto 0, r_0 \mapsto 0, r_1 \mapsto v_1, \ldots])\rangle$$

Then, $\Phi(\sigma_a) = \langle (l_0^a, [\underline{x} \mapsto v]), (l_2^a, [\underline{x} \mapsto 0])\rangle$.

The following definition introduces both the observational semantics of the program P and the operator that is used to compute it from $[\![P]\!]$:

Definition 3 (Observation operator). The *observational abstraction operator* α^r is $\widehat{\Phi}$. In other words, $\forall \mathcal{E} \in \mathbb{P}((L_P \times S_P)^\star)$, $\alpha^r(\mathcal{E}) = \{\Phi(t) \mid t \in \mathcal{E}\}$. The *observational semantics* $[\![P]\!]_r$ of the program P is defined by $[\![P]\!]_r = \alpha^r([\![P]\!])$.

As noted by the proposition below, the operator α^r is an abstraction operator:

Proposition 1 (Observation abstraction). *The operator α^r defines a Galois connection*

$$\mathbb{P}((L_P \times S_P)^\star) \underset{\alpha^r}{\overset{\gamma^r}{\leftrightarrows}} \mathbb{P}((L_P^r \times S_P^r)^\star)$$

Straightforward: We are in presence of complete lattices; α^r is monotone, hence, it determines uniquely the concretization operator γ^r so as to define a Galois connection. □

Intuitively, if $\mathcal{E} \in \mathbb{P}((L_P^r \times S_P^r)^\star)$, then $\gamma^r(\mathcal{E})$ denotes the set of all the traces σ of elements of $L_P \times S_P$ such that the restriction of σ belongs to $\mathcal{E}$.

3.3 Correctness of Compilation

In this subsection, we consider a source program P_s and an assembly program P_a. We suppose they are defined by two labeled transition systems $(L_s, V_s, i_s, \rightarrow_s)$ and $(L_a, V_a, i_a, \rightarrow_a)$. Our goal here is to formalize what a correct compilation of P_s into P_a is. We consider first the case of a simple compilation as opposed to an optimizing compilation. The case of more involved transformations is evoked afterwards.

We assume that we are given four sets $L_s^r \subseteq L_s$, $L_a^r \subseteq L_a$, $V_s^r \subseteq V_s$ and $V_a^r \subseteq V_a$ that define the program points and the memory locations of both programs which can be related (the notations $S_i^r = V_i^r \rightarrow R$ are also used in the following). More precisely, we suppose that two bijections $\pi_l : L_s^r \rightarrow L_a^r$ and $\pi_v : V_s^r \rightarrow V_a^r$ are defined. These bijections denote the correspondence between source and assembly program points and memory locations of both programs that are related as sketched in section 3.1:

. $\pi_v(x_s) = x_a$ expresses the fact that the memory location x_a stores a value equal to the value stored in x_s at corresponding program points (modulo a correspondence between source and assembly representation of data types, which we ignore here);
. π_v defines a store mapping $\pi_s : S_s^r \rightarrow S_a^r$: $\pi_s(\rho) = \rho \circ \pi_v^{-1}$;
. $\pi_l(l_s) = l_a$ means that a source state like (l_s, ρ_s) is related to an assembly state like (l_a, ρ_a) and conversely (where the correspondance between ρ_s and ρ_a is determined by π_s).

We can introduce a *trace mapping* operator now:

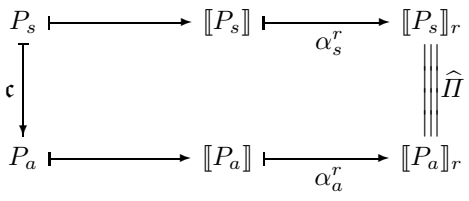
$$\begin{array}{rcl} \Pi : & (L_s^r \times S_s^r)^\star & \longrightarrow (L_a^r \times S_a^r)^\star \\ & \langle (l_0, \rho_0), \ldots, (l_n, \rho_n)\rangle & \longmapsto \langle (\pi_l(l_0), \pi_s(\rho_0)), \ldots, \\ & & \quad (\pi_l(l_n), \pi_s(\rho_n))\rangle \end{array}$$

The sets L_s^r and V_s^r (resp. L_a^r and V_a^r) define observation abstractions as in section 3.2. For instance, we write α_s^r (resp. γ_s^r) for the abstraction (resp. concretization) function associated to the definition of the observational semantics of source programs. The compilation of P_s into P_a is said to be correct if and only if the restricted semantics of both programs are in bijection:

Definition 4 (Correctness of compilation). The compilation $\mathfrak{c}$ of P_s into P_a is correct with respect to the mapping (π_l, π_v) if and only if the following holds:

$$\widehat{\Pi}([\![P_s]\!]_r) = [\![P_a]\!]_r$$

This situation can be depicted by the diagram below:



Example 2. We continue here the example of figure 3. The restricted sets are:

$$V_s^r = \{x\} \qquad L_s^r = L_s = \{l_0^s, l_1^s, l_2^s, l_3^s, l_4^s\}$$
$$V_a^r = \{\underline{x}\} \qquad L_a^r = \{l_0^a, l_2^a, l_6^a, l_{10}^a, l_{11}^a\}$$

The bijections π_l and π_v are defined on figure 3(c). The compilation of P_s into P_a is correct with respect to these mappings (in the sense of definition 4).

Furthermore, the assembly trace σ_a of example 1 is related to the following source trace σ_s (i.e. $\Pi(\sigma_s) = \sigma_a$):

$$\sigma_s = \langle (l_0^s, [x \mapsto v]), (l_1^s, [x \mapsto 0]) \rangle.$$

Extraction of the mappings π_v and π_l. In general the bijections π_v and π_l can be found in the output of the compiler. Indeed, most commonly used compilers provide auxiliary information for the sake of debugging. The mappings between program points and memory locations are among these "debugging information". Consequently, the use of these mappings should not be prohibitive in practice.

At the beginning of this subsection, we restricted to non-optimizing compilation; we give here a few hints about how to handle optimizations:

Remark 1 (Optimizations). Handling compiler optimizations generally requires to integrate it right at the compilation correctness definition level:

- As mentioned above, code or variable elimination based optimizations are handled by choosing π_s and π_l so as to get rid of the removed entities. So, definition 4 is general enough to deal with these optimizations.
- Many optimizations that change the structure of programs 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 iteration numbers and one for even iteration numbers. Handling this optimization reduces to splitting x into two program points x_{odd} and x_{even}. Hence, loop unrolling-based optimizations would require definition 4 to be extended to a more general definition that would allow the control structure of the source program to be unfolded.

Remark 2 (Practical variables mapping). In practice, the definition of the variable mapping π_v turns out to be more involved. Indeed, the source variables (hence, the source and assembly memory locations) have a restricted scope. Consequently, the relation between source and assembly memory locations depends on the program point. We assume in this paper that all the variables have a global scope and that π_v does not depend on the program point. Handling procedures requires solving this kind of technical issues.

The formalization of compilation presented above is equivalent to the approach of [ZPFG02]. It is also comparable to formalizations based on simulation techniques. However, we believe that the advantage of formalizing compilation inside the Abstract Interpretation framework is to bring both static analysis and compilation into a single framework, which makes reasoning about the process more simple, especially if we wish to extend it to optimizations. The observation abstractions considered in section 3.2 are simple projections; however, considering simple projection would not allow to generalize our presentation in order to deal with optimizations. Indeed, in the optimizing compilation case, the observations abstraction may have to be replaced by more complex operators (which would not be mere projections any more) and further developments would require to be extended accordingly.

4 Static Analysis and Invariant Translation

We consider now static analysis as a way of soundly approximating the possible behaviors of programs (more precisely, an abstract semantics is defined and then we compute a sound over-approximation of it). Then, we consider a "correct compilation" (in the sense of the previous section), and we show how to deduce abstract properties of the compiled program from abstract properties of the source program.

4.1 Abstract Domain and Static Analysis

We introduce here a class of static analyses, practically large enough to answer most questions of interest about the behavior of programs (like runtime errors detection). Let P be a program defined by a labeled transition system $(L_P, V_P, i_P, \rightarrow_P)$ (the corresponding set of stores is denoted by $S_P = V_P \rightarrow R$). We suppose that an abstract domain $D^\sharp$ is given for representing sets of stores:

$$(\mathbb{P}(S_P), \subseteq) \underset{\alpha^s}{\overset{\gamma^s}{\leftrightarrows}} (D^\sharp, \sqsubseteq)$$

The abstract semantics of a program is a function that maps a program point to the abstraction of the set of stores which can be encountered at this point in a trace of the program:

Definition 5 (Abstract semantics). The *trace abstraction* is defined as follows:

$$\alpha^t : \mathbb{P}((L_P \times S_P)^\star) \longrightarrow (L_P \rightarrow D^\sharp)$$
$$\forall \mathcal{E} \subseteq (L_P \times S_P)^\star,\ \forall l \in L_P,$$
$$\alpha^t(\mathcal{E})(l) = \alpha^s(\{\rho \mid \langle \ldots, (l, \rho), \ldots \rangle \in \mathcal{E}\})$$

The *abstract semantics* of P is defined by:

$$[\![P]\!]^\sharp = \alpha^t([\![P]\!])$$

The function α^t defines a Galois connection:

$$(\mathbb{P}((L_P \times S_P)^\star), \subseteq) \underset{\alpha^t}{\overset{\gamma^t}{\leftrightarrows}} (L_P \rightarrow D^\sharp, \dot{\sqsubseteq})$$

(same argument as in the case of proposition 1). □

Static analysis. In most cases, the abstract semantics $[\![P]\!]^\sharp$ is not exactly computable; hence we compute an over-approximation of it by using a sound *abstract semantic function* $F^\sharp_P : (L_P \to D^\sharp) \to (L_P \to D^\sharp)$ (the soundness of the abstract semantic function boils down to $\alpha^t \circ F_P \dot{\sqsubseteq} F^\sharp_P \circ \alpha^t$) and a widening operator ∇ in order to enforce convergence [CC79].

In the following we call *invariant* an element of the lattice $(L_P \to D^\sharp, \dot{\sqsubseteq})$. A *sound invariant* for the program P is an invariant I such that $[\![P]\!]^\sharp \dot{\sqsubseteq} I$; it provides a sound over-approximation of the set of reachable states of the program. Hence, static analysis computes a sound invariant for the program.

The definition of a sound abstract semantic function requires a few abstract operators to be introduced first. For instance, the two following abstract operators are sufficient to build an abstract semantic function for the programs written in the simple language of section 2.3 (the corresponding operators for the simple assembly language of section 2.4 will be designed in a very similar way in section 6.1):

- **Assignment**: the **assign** operator is defined by:

$$\mathbf{assign} : \mathtt{Lv} \times \mathtt{E} \times D^\sharp \longrightarrow D^\sharp$$

  Intuitively, it evaluates an l-value, an expression and operates the assignment in the abstract domain; in case the l-value does not evaluates into a single memory location but to a set of memory locations, the **assign** operator carries out a "may assign". The soundness of this operator can be stated as follows:

$$\begin{array}{l}\forall \rho \in S_P,\ \forall \rho^\sharp \in D^\sharp,\ \forall lv \in \mathtt{Lv},\ \forall e \in \mathtt{E},\\ \quad \rho \in \gamma^s(\rho^\sharp) \Longrightarrow \rho' \in \gamma^s(\mathbf{assign}(lv, e, \rho^\sharp))\end{array}$$

  where $\rho' = \rho[[\![lv]\!](\rho) \leftarrow [\![e]\!](\rho)]$ (the substitution operator also takes possible "may affect" into account).
- **Guard**: The **guard** operator is defined by:

$$\mathbf{guard} : \mathbb{B} \times \mathtt{C} \times D^\sharp \longrightarrow D^\sharp$$

  Intuitively, it inputs a boolean b, a condition c and an abstraction of a set of stores $\rho^\sharp$ and determines a superset of the stores abstracted by $\rho^\sharp$ such that c evaluates to b. Hence, the soundness of **guard** boils down to:

$$\begin{array}{l}\forall \rho \in S_P,\ \forall \rho^\sharp \in D^\sharp,\ \forall b \in \mathbb{B},\ \forall c \in \mathtt{C},\\ \quad (\rho \in [\![c]\!](b) \wedge \rho \in \gamma^s(\rho^\sharp)) \Longrightarrow \rho \in \gamma^s(\mathbf{guard}(b, c, \rho^\sharp))\end{array}$$

The abstract semantic function associated to the program P can be defined by considering the abstract transfer functions corresponding to all the statements in the program. More precisely, we write $\phi_{l,l'}$ for the abstract transfer function corresponding to the transition $l \to l'$. It should achieve the following soundness property:

$$\left.\begin{array}{l}\forall \rho, \rho' \in S_P,\ \forall \rho^\sharp \in D^\sharp,\\ \quad (l, \rho) \to_P (l', \rho')\\ \quad \rho \in \gamma^s(\rho^\sharp)\end{array}\right\} \Longrightarrow \rho' \in \gamma^s \circ \phi_{l,l'}(\rho^\sharp)$$

In case $\phi_{l,l'}$ is not defined explicitly below, then $\phi_{l,l'} = \lambda \rho^\sharp \in D^\sharp.\bot$:

- Case of an assignment $l : lv := e_1; l'$:

$$\phi_{l,l'}(\rho^\sharp) = \mathbf{assign}(lv, e, \rho^\sharp)$$

- Case of a conditional l : $\mathbf{if}(c)\,\{l_t : B_t; l'_t\}\,\mathbf{else}\,\{l_f : B_f; l'_f\};\ l' : \ldots$:

$$\begin{array}{l}\phi_{l,l_t}(\rho^\sharp) = \mathbf{guard}(\mathcal{T}, c, \rho^\sharp)\\ \phi_{l,l_f}(\rho^\sharp) = \mathbf{guard}(\mathcal{F}, c, \rho^\sharp)\\ \phi_{l'_t,l'}(\rho^\sharp) = \rho^\sharp\\ \phi_{l'_f,l'}(\rho^\sharp) = \rho^\sharp\end{array}$$

- Case of a loop l : $\mathbf{while}(c)\,\{l_b : B_b; l'_b\};\ l' : \ldots$:

$$\begin{array}{l}\phi_{l,l_b}(\rho^\sharp) = \mathbf{guard}(\mathcal{T}, c, \rho^\sharp)\\ \phi_{l,l'}(\rho^\sharp) = \mathbf{guard}(\mathcal{F}, c, \rho^\sharp)\\ \phi_{l'_b,l}(\rho^\sharp) = \rho^\sharp\end{array}$$

Fig. 4. Abstract semantic function $F^\sharp_P$

In other words, the abstract transfer function computes an over approximation of the set of stores at point l' which can be reached by following the transition $l \to l'$ starting from a given set of stores at point l. In case program traces cannot step from l to l', $\phi_{l,l'} = \lambda \rho^\sharp \in D^\sharp.\bot$.

The definition of all the abstract transfer functions for the program P proceeds by considering all the statements in the program as shown on figure 4.

The abstract semantic function mimics one execution step at the abstract level by applying the abstract transfer functions to the local invariants:

$$\begin{array}{l}F^\sharp_P : (L_P \to D^\sharp) \longrightarrow (L_P \to D^\sharp)\\ \text{if } l = i_P, \text{ then: } F^\sharp_P(I)(l) = \top\\ \text{if } l \neq i_P, \text{ then: } F^\sharp_P(I)(l) = \bigsqcup_{l' \in L_P} \phi_{l',l}(I(l'))\end{array}$$

This definition of $F^\sharp$ ensures soundness, since $\alpha^t \circ F_P \dot{\sqsubseteq} F^\sharp_P \circ \alpha^t$.

Extensions. The extension of such an analysis to a procedural language would not be difficult since it only requires to extend the notion of program point in order to enclose an execution stack. If recursion is not allowed (which is the case in many critical embedded systems), then the execution stack can be represented exactly at the abstract semantics level (a program point corresponds to a syntactic program point and a unique stack). On the contrary, if recursion is allowed, then an abstraction of sets of stacks must be defined in order to preserve the computer tractability (a label should represent a syntactic program point and a subset of the set of the stacks which may occur at this point).

In the following we will consider the case of the intervals domain only ($D^\sharp$ approximates the values of the

variables with intervals); however, the abstract domain $D^\sharp$ can be considered a parameter: It may be instantiated with other domains like affine equalities [Kar76], constants [Cou99] or octagons [Min01].

Last, we can remark that a control-based partitioning strategy similar to those described in [HT98]) can be used, in order to express more precise properties about programs. Then, a finite partition of the set of control paths ending in l is given for each program point and the abstract semantics of the program inputs both a program point and an element of the partition attached to this point and outputs an abstraction of the corresponding set of stores. This approach would require the extension of definition 5; however, this extension would be trivial.

Aspects of program certification. A large part of program certification consists in proving safety properties. For instance, the goal of runtime errors detection is to show that a program will not abort because of an illegal operation [BCC+02, BCC+03]. Program certification proceeds by checking that the set of concrete values represented by the abstract invariant cannot lead to an error (which is sound but obviously incomplete). For instance, if we discover an invariant $I \sqsupseteq [\![P]\!]^\sharp$ for a program P and in the case P contains the statement $l : A[i] := 10/v; l' : \ldots$ (where A is an array of size n_A and v a variable), we would have to check:

- the correctness of the array assignment: $\forall \rho \in \gamma^s(I(l)),\ 0 \leq \rho(i) < n_A$ (no out-of-bound array access);
- the correctness of the division: $\forall \rho \in \gamma^s(I(l)),\ 0 \notin \rho(v)$ (no divide by 0 error).

An example analysis. Given $S_P = V_P \to \mathbb{Z}^o$, we envisage here a simple interval analysis. The Galois connection $(\mathbb{P}(S_P), \subseteq) \underset{\alpha^s}{\overset{\gamma^s}{\leftrightarrows}} (D^\sharp, \sqsubseteq)$ is defined by:

$$D^\sharp = V_P \to (\{\bot\} \cup \{[a, b] \mid a, b \in \mathbb{Z}^o,\ a \leq b\})$$

and:

$$\begin{array}{l}\alpha^s(\emptyset) = \bot \\ \forall \mathcal{E} \subseteq S_P \text{ such that } \mathcal{E} \neq \emptyset,\ \forall x \in V_P, \\ \qquad \alpha^s(\mathcal{E})(x) = [\min\{\rho(x) \mid \rho \in \mathcal{E}\}, \max\{\rho(x) \mid \rho \in \mathcal{E}\}]\end{array}$$

$$\begin{array}{l}\gamma^s(\bot) = \emptyset \\ \forall \rho^\sharp \in D^\sharp,\ \gamma^s(\rho^\sharp) = \\ \qquad \{(\lambda x \in V_P . v_x) \mid \forall x \in V_P,\ x_{\min} \leq v_x \leq x_{\max} \\ \qquad\qquad \text{where } \rho^\sharp(x) = [x_{\min}, x_{\max}]\}\end{array}$$

The transfer functions and the complete domain definition are trivial and can be found in [Cou99].

Example 3. The most precise sound invariant for program P_c is displayed in the table below. One can remark that a simple interval analyzer would discover this invariant exactly.

Program point l	$[\![P]\!]^\sharp(l)(x)$
l_0^s	$[N_{\min}, N_{\max}]$
l_1^s	$[0, 100]$
l_2^s	$[0, 99]$
l_3^s	$[1, 100]$
l_4^s	$[100, 100]$

4.2 Invariant Translation

In this subsection, we use the same notations for a source program P_s and for an assembly program P_a as we did in section 3.3. The compilation of P_s into P_a is supposed correct in the sense of definition 4 (the mappings for the program points, the variables and the stores π_l, π_v and π_s are also defined in the same way as in section 3.3). Furthermore, we assume that $V_s^r = V_s$ for the sake of simplicity (the general case will be treated in the following subsections). We also make the assumption that an abstract interpretation is defined for the source program P_s: $D_s^\sharp$ denotes the abstract domain for representing sets of source stores (the corresponding Galois connection is defined by the pair of functions (α_s^s, γ_s^s)). Moreover, we assume that $I_s \in (L_s \to D_s^\sharp)$ is a sound invariant for the source program (i.e: $[\![P_s]\!]^\sharp \dot{\sqsubseteq} I_s$). We write Φ_s and Φ_a for the trace restriction operators (defined as in section 3.2). The store restriction operators are denoted by ϕ_s and ϕ_a.

Let $l_s \in L_s^r$ and $l_a = \pi_l(l_s)$.

Let $\sigma_a = \langle \ldots, (l_a, \rho_a), \ldots \rangle \in [\![P_a]\!]$. The correctness of the compilation of P_s into P_a entails that $\widehat{\Pi}([\![P_s]\!]_r) = [\![P_a]\!]_r$; consequently, there exists a trace $\sigma_s \in [\![P_s]\!]$ such that $\Pi(\Phi_s(\sigma_s)) = \Phi_a(\sigma_a)$. Since $l_a \in L_a^r$, σ_s is of the form

$$\sigma_s = \langle \ldots, (l_s, \rho_s), \ldots \rangle$$

and $\phi_a(\rho_a) = \pi_s(\phi_s(\rho_s))$.

Hence, $\rho_s = \phi_s(\rho_s) = \phi_a(\rho_a) \circ \pi_v$.

The soundness of the invariant I_s entails that $\rho_s \in \gamma_s^s(I_s(l_s))$. Consequently, $\phi_a(\rho_a) \circ \pi_v \in \gamma_s^s(I_s(l_s))$.

At this point we have shown the proposition:

$$\left.\begin{array}{l}[\![P_s]\!]^\sharp \dot{\sqsubseteq} I_s \\ l_a = \pi_l(l_s) \\ \langle \ldots, (l_a, \rho_a), \ldots \rangle \in [\![P_a]\!]\end{array}\right\} \Longrightarrow \phi_a(\rho_a) \circ \pi_v \in \gamma_s^s(I_s(l_s)).$$

This proposition sketches how an abstract property for the assembly program can be derived from an abstract invariant of the source program, even if it does not lead to an explicit soundness statement like $\rho_a \in \gamma_a^s(\rho_a^\sharp)$ where $\rho_a^\sharp$ is an element of a suitable abstract domain:

$$(\mathbb{P}(V_a \to R), \subseteq) \underset{\alpha_a^s}{\overset{\gamma_a^s}{\leftrightarrows}} (D_a^\sharp, \sqsubseteq)$$

In the following, the design of a translated invariant is done in two steps: A "restricted abstract semantics" is first defined, which is both an abstraction of the observational semantics of section 3.2 and of the abstract semantics underlying static analysis (section 4.1); then the

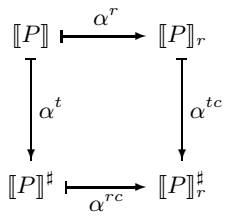


Fig. 5. Abstractions of the standard semantics

translator is defined as a function which inputs a source restricted abstract semantics and outputs an assembly restricted abstract semantics.

4.3 Abstract Semantics and Observation

In this subsection, we use the same notations as in section 4.1: We consider a program P and we suppose that an abstract interpretation of the sets of stores of P is given and extended to an abstract semantics for P. The purpose of this subsection is to show how a new abstract semantics can be defined in order to represent sets of projected stores (i.e. subsets of $V_P^r \to R$) as shown on figure 5. We describe here an effective way to define $[\![P]\!]^\sharp_r$, α^{tc} and α^{rc}.

Forget operator. An operator $\mathbf{forget} : V_P \times D^\sharp \to D^\sharp$ inputs a variable $x \in V_P$ and an abstract value $\rho^\sharp$ and outputs a new abstract value $\rho'^\sharp$ which does not take into account the variable x by forgetting about any information concerning to this variable. The soundness of a **forget** operator is stated as follows:

$$\begin{array}{l}\forall x \in V_P,\ \forall \rho, \rho' \in S_P,\ \forall \rho^\sharp \in D^\sharp,\\ \quad (\forall y \in V_P,\ y \neq x \Longrightarrow \rho(y) = \rho'(y)) \Longrightarrow\\ \qquad \rho \in \gamma^s(\rho^\sharp) \Longrightarrow \rho' \in \gamma^s(\mathbf{forget}(x, \rho^\sharp))\end{array}$$

Furthermore, a forget operator should be idempotent:

$$\begin{array}{l}\forall x \in V_P,\ \forall \rho^\sharp \in D^\sharp,\\ \quad \mathbf{forget}(x, \mathbf{forget}(x, \rho^\sharp)) = \mathbf{forget}(x, \rho^\sharp).\end{array}$$

Indeed, forgetting twice about the constraints on variable x yields the same result as forgetting about them only once.

The definition of such an operator is trivial for most domains: $\mathbf{forget}(x, \rho^\sharp)$ basically removes all the constraints on variable x. In most cases (domains of intervals, octagons, linear equalities...), the **forget** operator achieves the following exactness condition (the domain of polyhedra achieves a similar property despite it does not enjoy an abstraction function):

$$\begin{array}{l}\forall x \in V_P,\ \forall \mathcal{X} \subseteq S_P,\\ \quad \mathbf{forget}(x, \alpha^s(\mathcal{X})) =\\ \qquad \alpha^s(\{\rho' \in S_P \mid \exists \rho \in \mathcal{X},\\ \qquad\quad \forall y \in V_P,\ y \neq x \Rightarrow \rho(y) = \rho'(y)\})\end{array}$$

In the following, all the forget operators we consider are supposed to be exact and idempotent.

Such an operator can be straightforwardly extended to an operator on sets of variables (forgetting about a set of variables amounts to forgetting about all of them in any order).

Last, note that $\mathbf{forget}(V_P \setminus V_P^r, \rho^\sharp)$ in fact corresponds to an element of a "restricted abstract domain" $D^{r\sharp}$ which defines a Galois connection with $\mathbb{P}(S_P^r)$:

$$(\mathbb{P}(S_P^r), \subseteq) \underset{\alpha^{sr}}{\overset{\gamma^{sr}}{\leftrightarrows}} (D^{r\sharp}, \sqsubseteq)$$

This domain can be defined as follows:

Definition 6 (Restricted abstract domain). The *restricted abstract domain* is defined by:

$$D^{r\sharp} = \{\mathbf{forget}(V_P \setminus V_P^r, \rho^\sharp) \mid \rho^\sharp \in D^\sharp\}$$

Moreover, if $X \in \mathbb{P}(S_P^r)$,

$$\alpha^{sr}(X) = \mathbf{forget}(V_P \setminus V_P^r, \alpha^s(\{\rho \in S_P \mid \phi(\rho) \in X\}))$$

For instance, in the case of the interval domain seen in section 4.1, $\mathbf{forget}(V_P \setminus V_P^r, \rho^\sharp)$ is a function from V_P to intervals which maps any variable that does not belong to V_P^r to the "top" interval $[N_{\min}, N_{\max}]$. Consequently, an abstract value of $D^{r\sharp}$ is isomorphic to a function from V_P^r to intervals.

Towards a more adapted abstract semantics. At this point a new abstract semantics can be defined, which takes into account the variables of V_P^r only (and implements the diagram of figure 5):

Definition 7 (Restricted abstract semantics). Let α^{rc} be the function defined by:

$$\begin{array}{rcl}\alpha^{rc} : (L_P \to D^\sharp) & \longrightarrow & (L_P^r \to D^{r\sharp})\\ I & \longmapsto & \lambda l \in L_P^r.(\mathbf{forget}(V_P \setminus V_P^r, I(l)))\end{array}$$

The *restricted abstract semantics* $[\![P]\!]^\sharp_r$ of the program P is defined by

$$[\![P]\!]^\sharp_r = \alpha^{rc}([\![P]\!]^\sharp)$$

As shown by the (trivial) following proposition, the restricted abstract semantics is an abstraction of the "standard" abstract semantics $[\![P]\!]^\sharp$:

Proposition 2. *The function α^{rc} is the abstraction function of a Galois connection*

$$(L_P \to D^\sharp, \dot{\sqsubseteq}) \underset{\alpha^{rc}}{\overset{\gamma^{rc}}{\leftrightarrows}} (L_P^r \to D^{r\sharp}, \dot{\sqsubseteq})$$

Straightforward. □

Observation and restricted abstract semantics. The restricted abstract semantics $[\![P]\!]^\sharp_r$ can also be seen as an abstraction of the observational semantics introduced in section 3.2 in order to define the correctness of compilation:

Proposition 3. *Let α^{tc} be the function defined by analogy with α^t by:*

$$\begin{array}{l}\alpha^{tc} : \mathbb{P}((L^r_P \times S^r_P)^\star) \longrightarrow (L^r_P \to D^{r\sharp}) \\ \forall \mathcal{E} \subseteq (L^r_P \times S^r_P)^\star,\ \forall l \in L^r_P, \\ \quad \alpha^{tc}(\mathcal{E})(l) = \alpha^{sr}(\{\rho \mid \langle \ldots, (l,\rho), \ldots\rangle \in \mathcal{E}\})\end{array}$$

This function is the abstraction function of a Galois connection

$$(\mathbb{P}((L^r_P \times S^r_P)^\star), \subseteq) \underset{\alpha^{tc}}{\overset{\gamma^{tc}}{\leftrightarrows}} (L^r_P \to D^{r\sharp}, \dot{\sqsubseteq})$$

Furthermore,

$$\alpha^{tc} \circ \alpha^r = \alpha^{rc} \circ \alpha^t$$

Hence, $[\![P]\!]^\sharp_r = \alpha^{tc}([\![P]\!]_r)$.

Let $\mathcal{E} \subseteq (L_P \times S_P)^\star$ and $l \in L^r_P$.

$$\begin{array}{l}\alpha^{tc} \circ \alpha^r(\mathcal{E})(l) \\ \quad = \alpha^{sr}(\{\rho \mid \langle \ldots, (l,\rho), \ldots\rangle \in \alpha^r(\mathcal{E})\}) \\ \quad = \alpha^{sr}(\{\rho \mid \langle \ldots, (l,\rho), \ldots\rangle \in \{\Phi(\sigma) \mid \sigma \in \mathcal{E}\}\}) \\ \quad = \alpha^{sr}(\{\phi(\rho) \mid \langle \ldots, (l,\rho), \ldots\rangle \in \mathcal{E}\}) \\ \quad = \textbf{forget}(V_P \setminus V^r_P, \\ \qquad \alpha^s(\{\rho \mid \phi(\rho) \in \{\phi(\rho) \mid \langle \ldots, (l,\rho), \ldots\rangle \in \mathcal{E}\}\})) \\ \quad = \textbf{forget}(V_P \setminus V^r_P, \\ \qquad \alpha^s(\{\rho \mid \exists \rho' \in S_P,\ \langle \ldots, (l,\rho'), \ldots\rangle \in \mathcal{E} \\ \qquad\qquad \wedge \forall y \in V^r_P,\ \rho(y) = \rho'(y)\})) \\ \quad = \textbf{forget}(V_P \setminus V^r_P, \\ \qquad \textbf{forget}(V_P \setminus V^r_P, \\ \qquad\quad \alpha^s(\{\rho \in S_P \mid \langle \ldots, (l,\rho'), \ldots\rangle \in \mathcal{E}\}))) \\ \qquad \text{since } \textbf{forget} \text{ is exact} \\ \quad = \textbf{forget}(V_P \setminus V^r_P, \alpha^s(\{\rho \mid \langle \ldots, (l,\rho), \ldots\rangle \in \mathcal{E}\})) \\ \qquad \text{since } \textbf{forget} \text{ is idempotent} \\ \quad = \alpha^{rc} \circ \alpha^t(\mathcal{E})(l)\end{array}$$

□

At this point, we have introduced three abstractions of the standard concrete semantics $[\![P]\!]$, shown on the diagram of figure 5:

- $[\![P]\!]_r$ is the observational semantics (correctness of compilation is expressed with respect to it);
- $[\![P]\!]^\sharp$ underlies static analysis;
- $[\![P]\!]^\sharp_r$ is an abstraction of these two semantics (intuitively, the dual of a reduced product).

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.

4.4 Invariant Translation Correctness

In this section, we consider a source program P_s and an assembly program P_a as in section 4.2. We assume that the compilation of P_s into P_a is correct, hence π_v, π_s, π_l and Π are defined as in section 3.3. All the notations of section 4.2 apply; however, the assumption that $V^r_s = V_s$ is relaxed. An abstract domain $D^\sharp_s$ (resp. $D^\sharp_a$) is defined for the source program (resp. the assembly program). The link between both domains is made explicit in the following. A **forget** operator is defined both at the source and at the assembly level; furthermore, restricted abstract domains are defined as in the previous subsection and are denoted by $D^{r\sharp}_s$ and $D^{r\sharp}_a$.

Invariant translation. An invariant translation procedure is based on a function which maps an abstract value $\rho^\sharp_s \in D^{r\sharp}_s$ to an abstract value $\rho^\sharp_a \in D^{r\sharp}_a$, and which is an abstract counterpart for π_s. Let $\pi^\sharp_s$ be such a function.

Definition 8 (Sound abstract translation operator). The abstract store translation function $\pi^\sharp_s$ is *sound* if and only if $\alpha^{sr}_a \circ \widehat{\pi}_s \dot{\sqsubseteq} \pi^\sharp_s \circ \alpha^{sr}_s$.

Furthermore, $\pi^\sharp_s$ is *exact* if and only if $\alpha^{sr}_a \circ \widehat{\pi}_s = \pi^\sharp_s \circ \alpha^{sr}_s$.

The notion of sound abstract translation operator introduced in definition 8 operates the invariant translation sketched in section 4.2: If $\rho^\sharp$ is a sound approximation of a set $\mathcal{E}$ of source restricted stores, then $\pi^\sharp_s(\rho^\sharp)$ is a sound approximation of the set of assembly restricted stores $\widehat{\pi}_s(\mathcal{E})$.

Once an abstract translation operator $\pi^\sharp_s$ is given, an abstract invariant translation operator $\Pi^\sharp$ can be defined as follows:

$$\begin{array}{rcl}\Pi^\sharp : (L^r_s \to D^{r\sharp}_s) & \longrightarrow & (L^r_a \to D^{r\sharp}_a) \\ I & \longmapsto & \lambda l_a \in L^r_a.(\pi^\sharp_s(I(\pi^{-1}_l(l_a))))\end{array}$$

If $\pi^\sharp_s$ is sound, then $\alpha^{tc}_a \circ \widehat{\Pi} \dot{\sqsubseteq} \Pi^\sharp \circ \alpha^{tc}_s$. In this case, we say that $\Pi^\sharp$ is sound. Similarly, if $\pi^\sharp_s$ is exact, then the equality holds and $\Pi^\sharp$ is said to be exact.

The soundness of an abstract translation operator is defined with respect to a correct compilation since it involves the mapping operators π_v and π_l.

In practice, the domains $D^{r\sharp}_s$ and $D^{r\sharp}_a$ are similar: If the first is an interval domain, then so is the second. Hence, the design of a sound abstract translation operator is straightforward and completely guided by the variables mapping π_v. Moreover, $\pi^\sharp_s$ is always exact and monotone in practice. The definition of $\Pi^\sharp$ is also completely determined by the translation information since it is based on the function $\pi^\sharp_s$ and on the program points mapping π_l.

Correctness. At this point, we can state the soundness of the method: If we use a sound source analyzer, a correct compiler and a sound invariant translator, the process yields a safe invariant for the compiled program.

Theorem 1 (Invariant translation correctness). *If the compilation of P_s into P_a is correct, if $\pi^\sharp_s$ is sound and monotone and if $I_s \in (L_s \to D^\sharp_s)$ is a sound invariant for P_s (i.e. $[\![P_s]\!]^\sharp \dot{\sqsubseteq} I_s$), then $I^r_a = \Pi^\sharp \circ \alpha^{rc}_s(I_s)$ is a sound restricted abstract invariant for the assembly program, that is $[\![P_a]\!]^\sharp_r \dot{\sqsubseteq} I^r_a$.*

Hence, a sound abstract invariant for the assembly program can be derived:

$$[\![P_a]\!]^\sharp \dot{\sqsubseteq} \gamma^{rc}_a \circ \Pi^\sharp \circ \alpha^{rc}_s(I_s)$$

We first choose $I_s = [\![P_s]\!]^\sharp$; this situation is illustrated in figure 6. Then:

$$\begin{aligned} \Pi^\sharp \circ \alpha^{rc}_s(I_s) &= \Pi^\sharp \circ \alpha^{rc}_s \circ \alpha^t_s([\![P_s]\!]) \\ &= \Pi^\sharp \circ \alpha^{tc}_s \circ \alpha^r_s([\![P_s]\!]) \text{ (proposition 3)} \\ &= \Pi^\sharp \circ \alpha^{tc}_s([\![P_s]\!]_r) \end{aligned}$$

and:

$$\begin{aligned} [\![P_a]\!]^\sharp_r &= \alpha^{tc}_a([\![P_a]\!]_r) \qquad \text{(proposition 3)} \\ &= \alpha^{tc}_a \circ \widehat{\Pi}([\![P_s]\!]_r) \end{aligned}$$

since the compilation is correct ($[\![P_a]\!]_r = \widehat{\Pi}([\![P_s]\!]_r)$).

The soundness of $\Pi^\sharp$ entails that:

$$[\![P_a]\!]^\sharp_r \dot{\sqsubseteq} \Pi^\sharp \circ \alpha^{rc}_s(I_s)$$

In general, $[\![P_s]\!]^\sharp \dot{\sqsubseteq} I_s$. Given α^{rc}_s and $\pi^\sharp_s$ are monotone, the assembly invariant $\Pi^\sharp \circ \alpha^{rc}_s(I_s)$ is a sound approximation of $[\![P_a]\!]^\sharp_r$. □

We mentioned in section 3.3 that allowing further compiler optimizations and transformations would require the observational semantics $[\![.]\!]_r$ to be adapted. This would entail the extension of the semantics $[\![.]\!]^\sharp_r$ and of the invariant translation procedure $\pi^\sharp_s$. However, the methodology we presented in this section is general and would not be changed.

Example 4. The source invariant displayed in the example 3 can be translated into the assembly invariant I^r_a given in the table below. Note that this invariant is a sound approximation of the abstract semantics of the assembly program as proved by theorem 1 ($[\![P_a]\!]^\sharp_r \dot{\sqsubseteq} I^r_a$).

Program point l	$I^r_a(l)(\underline{x})$
l^a_0	$[N_{\min}, N_{\max}]$
l^a_2	$[0, 100]$
l^a_6	$[0, 99]$
l^a_{10}	$[1, 100]$
l^a_{11}	$[100, 100]$

Towards a more informative and safer invariant. The invariant translation procedure defined above does not yield a very accurate invariant for the assembly program. Roughly speaking, the invariant $I_a = \gamma^{rc}_a(I^r_a)$ does not tell us anything about the value of the memory locations that do not belong to V^r_a or about the value of any variable at a program point $l \notin L^r_a$. Furthermore, the correctness of the translated invariant relies on several assumptions which we would like to relax (the compiler, the invariant translator and the source invariant are supposed to be correct). Therefore, we show in the following how to construct a "full" invariant for the assembly program (i.e. that would tell something about all the assembly memory locations and program points) and how to check it independently.

5 Invariant Propagation and Invariant Checking

This section exposes how to carry out the invariant propagation (i.e. computation of a more precise invariant for the assembly program) and the invariant checking, which should allow to trust the assembly invariant apart from any hypothesis about the compilation, the source analysis or the invariant translation.

5.1 Post-fixpoints and Post-iterations

In this section we keep the notations of section 4.4. In particular, I_a denotes the translated invariant which we would like to refine and to check.

Assembly invariant checker. In section 4.1, a static analysis was defined for a program P by giving a computable function $F^\sharp_P : (L_P \to D^\sharp) \to (L_P \to D^\sharp)$ such that $\alpha^t \circ F_P \dot{\sqsubseteq} F^\sharp_P \circ \alpha^t$ (soundness of the abstract semantic function $F^\sharp_P$) where F_P denotes the concrete semantic function introduced in definition 2. Such a function $F^\sharp_a : (L_a \to D^\sharp_a) \to (L_a \to D^\sharp_a)$ can be defined for the assembly program (with the same soundness condition with respect to the concrete semantic function of the assembly program F_a), as will be done in section 6.1. The soundness of $F^\sharp_a$ entails that $[\![P_a]\!]^\sharp \dot{\sqsubseteq} \mathrm{lfp} F^\sharp_a$. In practice, $F^\sharp_a$ is monotone. Note that this function could define an analyzer for the assembly program as shown in section 4.1 in the case of the program P: A sound approximation of $[\![P_a]\!]^\sharp$ could be computed by iterating this function from $\bot$ and using a widening operator and an appropriate iteration strategy [Bou93]. However, the lack of information available at the assembly level would make the design of an efficient iteration strategy rather involved as mentioned in the introduction.

Post-fixpoint and invariant checking. The invariant I_a is said to be a post-fixpoint of $F^\sharp_a$ if and only if $F^\sharp_a(I_a) \dot{\sqsubseteq} I_a$. In case I_a is a post-fixpoint of $F^\sharp_a$, then $\mathrm{lfp} F^\sharp_a \dot{\sqsubseteq} I_a$, therefore I_a is a sound approximation of $[\![P_a]\!]^\sharp$.

Therefore the checking of a candidate assembly invariant can be done just by verifying that it is a post-fixpoint of the assembly abstract semantic function $F^\sharp_a$. If the translated invariant is a post-fixpoint, then it is sound apart from any hypothesis concerning the way it

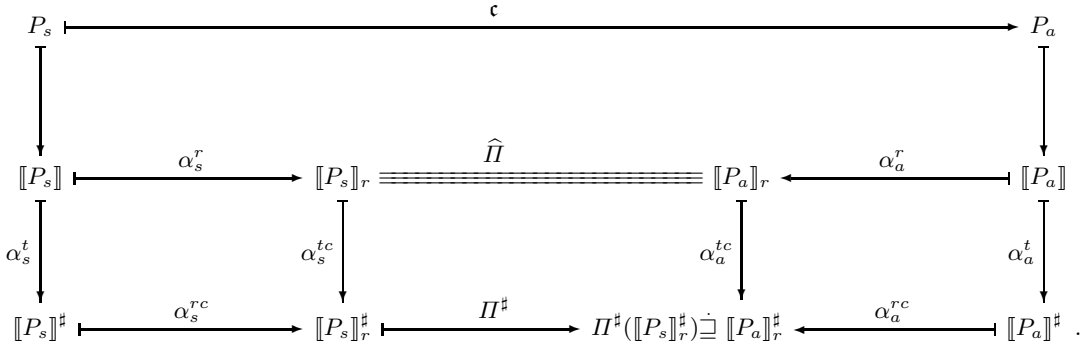


Fig. 6. Proof of theorem 1

was obtained (the source analyzer, the invariant translator and the compiler are not required to be sound any more to trust the translated invariant as was done in theorem 1). However, in case I_a is not a post-fixpoint of $F^\sharp_a$, we cannot conclude it is not a sound approximation of $[\![P_a]\!]^\sharp$.

In practice, the assembly abstract domain and the assembly transfer functions should be defined carefully so as to make the checking possible. Moreover, the refinement of the invariant (invariant propagation) should be done before the invariant checking.

Post-iteration(s) and invariant propagation. If I is a post-fixpoint of $F^\sharp_a$, then the sequence $(I_n)_{n\in\mathbb{N}}$ defined by $I_0 = I$ and $I_{n+1} = F^\sharp_a(I_n)$ is decreasing since $F^\sharp_a$ is monotone. Hence, a way of improving the precision of the translated invariant is to iterate the assembly abstract transfer function starting from I_a if it is a post-fixpoint.

In case I_a is not a post-fixpoint, then an iteration sequence can still be computed starting from it. However, a widening operator would generally be necessary to enforce convergence.

Nevertheless the translated invariant is generally not a post-fixpoint: I_a maps elements of L^r_a to precise abstractions of sets of stores but it maps the elements of $L_a \backslash L^r_a$ to the least precise local invariant $\top$, which makes the checking unsuccessful at the points of L^r_a. Therefore, the next subsection explains how to compute a post-fixpoint I'_a from the translated invariant I_a. Then I'_a can be checked as mentioned above.

5.2 Practical Solution

In practice, the program point mapping π_l maps at least one point in each loop of the assembly control flow graph to a source program point. Therefore, the computation of a post-fixpoint of $F^\sharp_a$ does not require an unbounded iteration.

We suppose now that I_a is sound. Let $l \in L_a \setminus L^r_a$. Then, a sound local invariant can be determined for this point by considering all the paths from a point in L^r_a to l which do not encounter another point belonging to L^r_a. Indeed, given such a path $c = l', l_0, \ldots, l_n, l$ where $l' \in L^r_a$, we can compute a sound abstract approximation I^c_l of the set of stores $\{\rho \mid \langle \ldots, (l', \rho'), (l_0, \rho_0), \ldots, (l_n, \rho_n), (l, \rho)\rangle \in [\![P_a]\!]\}$ by using the abstract transfer functions introduced in section 4.1:

$$I^c_l = \bigsqcup\{\ \phi_{l_n,l} \circ \phi_{l_{n-1},l_n} \circ \ldots \circ \phi_{l_0,l_1} \circ \phi_{l',l_0}(I_a(l_0)) \mid (\forall i,\ l_i \notin L^r_a) \wedge (l' \in L^r_a)\}$$

This amounts to iterating $F^\sharp_a$ from the abstract element $J_a : L_a \to D^\sharp_a$ displayed below:

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

Then, if N is the maximal length of a path $c = l', l_0, \ldots, l_n$ such that $l' \in L^r_a$ and $\forall i,\ l_i \in L_a \setminus L^r_a$, a sound invariant I_a can be computed in N iterations. Furthermore, this invariant would provide precise information for any point of the control flow graph of P_a:

$$I'_a = \bigsqcup_{i=0}^{N} (F^\sharp_a)^i(J_a).$$

In practice, I'_a is adapted to invariant checking. Furthermore, checking that I'_a is a post-fixpoint for $F^\sharp_a$ reduces to showing the following local property for each pair $(l, l') \in (L_a \setminus L^r_a) \times L^r_a$:

$$\phi_{l,l'}(I'_a(l)) \sqsubseteq I'_a(l')$$

(indeed, the local invariants at all the other points of the graph have been computed so as to achieve this property).

5.3 Incompleteness

Section 5.2 details a method which should lead to the checking of an invariant I'_a (by verifying it is a post-fixpoint of $F^\sharp_a$) derived from the translated invariant I_a.

However, the invariant checking is definitely incomplete. For instance, I'_a may fail to be a post-fixpoint of $F^\sharp_a$

if the abstract domain for the assembly program is too weak to express some intermediate properties necessary for the checking to succeed or if the abstract transfer function is not precise enough.

Intuitively, a very simple piece of source code may be compiled into a very obfuscated piece of target code. In case the source statement is simply a "skip" statement, the assembly checker would have to check that the corresponding piece of code does not modify the abstract stores. Nevertheless, the fact that a piece of code "does nothing" is not decidable and so is in general the fact that a piece of code "does nothing at the abstract level".

In practice, the implementation of the method starts by the definition of a class of source and compiled programs we wish the checking to succeed for, which amounts to choosing the features of the source language allowed and a class of compilers and compilation options. Then comes the choice of the assembly abstract domain (which may need to be obtained by refining the source abstract domain in order to convey "more" intermediate properties) and of the abstract transfer functions for assembly programs. This step is crucial and should lead to the automatic checking of the programs of the previously defined class, following the method proposed in section 5.2. We believe it is generally possible to build a "good" abstract domain for a large class of source and compiled programs. For instance, in the case of our experiment based on a significant subset of the C language and on the PowerPC architecture, only three refinements of the domain were required. Two of them are due to the way conditional branching is done in assembly programs and are described in detail in section 6 (the third one is evoked briefly in section 7). These refinements were made necessary by particular aspects of the assembly language: They should be handled only once even if we wish to use several compilers since the refinements are not specific to the compiler but to the assembly language.

The purpose of designing a tool which would be "complete on a class of programs" does not contradict the incompleteness of the method. Indeed, given a tool which is complete on the class of programs we are interested in, it is generally possible to design a source program P_s and an assembly program P_a (outside of the class) such that the compilation of P_s into P_a is correct in the sense of definition 4 and such that the checking of a translated invariant computed from an invariant obtained on P_s fails.

Anyway, a failure at checking time should lead to the manual inspection of the cause. In case the failure is due to a weakness of the abstract domain, the domain should be improved.

The case of compiler optimizations (not considered in this paper) turns out to be similar. Indeed, the choice the optimizations to allow is part of the first step (definition of a class of source and compiled programs). The abstract domain and transfer functions still should be chosen accordingly.

5.4 An Abstract Proof of Compilation

When the checking of an invariant I'_a succeeds on an assembly program P_a, the invariant I'_a can be considered sound apart from any hypothesis about the compilation of P_s into P_a or about the way the invariant I_a was produced. Then I_a provides information about the behavior of the assembly program P_a: For instance, the value stored in the memory cell of address $\underline{x}$ is in the range $[1, 100]$ at the program point l^a_{10} in the assembly program of figure 3(b) (the propagation and checking of the invariant displayed in the example 4 will be described formally in the next section). However the correctness of the compilation itself is not proved: In the example, the checking of the invariant does not prove that the value stored in the memory cell of address $\underline{x}$ at point l^a_{10} is the equal to the value of variable x at point l^s_3 in the source program, even if it shows that these values both belong to the range $[1, 100]$.

However, the assembly level checking of an invariant that was derived from a source invariant provides a kind of "abstract proof of compilation": Indeed, it entails that the compiled program does not present behaviors the source analyzer proved that the source program does not enjoy. Therefore, this approach may detect *some* bugs of compilers whereas other bugs cannot be detected. By contrast, Translation Validation [PSS98] aims at proving an operational equivalence between source and target programs. Consequently, this method should be more adapted to the discovery of compiler bugs.

6 Practical Aspects of Invariant Propagation and Invariant Checking

Previous sections gave an overview of invariant translation and invariant checking. Given the method is not complete (checking may fail even if the translated invariant is sound), the design of a precise abstract domain is required for checking to succeed. We envisage common refinements, which turned out to be necessary for a specific (yet representative) architecture.

6.1 Definition of the Assembly-Level Abstract Checker

In this section we are interested in the checking of invariants on programs produced by simple non optimizing compilers for the target architecture described in section 2.5 (which defines the class of compiled programs we are interested in). This includes the **gcc** compiler for the PowerPC architecture with most optimizations turned off: The prototype presented in section 7 was designed

for this architecture and this compiler; hence, it basically implements the domain presented in this section.

The invariant checking method presented in section 5 is based on an abstract semantic function $F_a^\sharp$ for the assembly program P_a. We assume here that an assembly domain $D_0^\sharp$ is given together with a Galois connection $(\mathbb{P}(S_a), \subseteq) \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} (D_0^\sharp, \sqsubseteq)$ and we define a sound abstract semantic function $F_a^\sharp : (L_a \to D_0^\sharp) \longrightarrow (L_a \to D_0^\sharp)$ for P_a; in the following, we show how to instantiate $D_0^\sharp$ so as to make checking succeed on the class of programs under consideration.

As in section 4.1, we suppose that $D_0^\sharp$ provides two abstract operators to handle assignments and tests:

- **assign** : $L \times E \times D_0^\sharp \longrightarrow D_0^\sharp$ where L denotes the set of assembly l-values (including all the registers and the expressions that define one or several memory cells) and E denotes the set of the expressions depending on the content of assembly memory cells.
- **guard** : $\mathbb{B} \times C \times D_0^\sharp \longrightarrow D_0^\sharp$ where C denotes the set of the conditional expressions depending on the content of assembly memory cells.

The definition of the assembly abstract function $F_a^\sharp$ is also based on abstract transfer functions:

$$F_a^\sharp : (L_a \to D_0^\sharp) \longrightarrow (L_a \to D_0^\sharp)$$

If $I_a \in (L_a \to D_0^\sharp)$, then:

$$\begin{array}{l} \text{if } l = i_a, \text{ then: } F_a^\sharp(I_a)(l) = \top \\ \text{if } l \neq i_a, \text{ then: } F_a^\sharp(I_a)(l) = \bigsqcup_{l' \in L_P} \phi_{l',l}(I_a(l')) \end{array}$$

Intuitively, $\phi_{l,l'}$ defines the abstract transition from point l to point l'. The abstract transfer functions are defined in detail in figure 7. The soundness of $F_a^\sharp$ boils down to the soundness of the transfer functions in the same way as in section 4.1.

The abstract domain used for computing an invariant for the source program is the domain of intervals (section 4.1), so the first choice for $D_0^\sharp$ is also based on intervals. Yet, variables values in assembly programs not only include integers but also condition register values; therefore $D_0^\sharp$ also relies on the domain of constants $D_\mathbb{C}$ defined by $\mathbb{C}$. Another possible choice would be to use a domain based on $\mathbb{P}(\mathbb{C})$; the results would be a little more precise, yet the problems mentioned in the following subsection would still occur. In practice, the condition register is the only memory cell that may contain a value in $\mathbb{C}$, which justifies the following choice for $D_0^\sharp$:

$$\begin{array}{rl} D_0^\sharp &= (\{\mathrm{cr}\} \to D_\mathbb{C} \times ((L_a \setminus \{\mathrm{cr}\}) \to \mathbb{I}_{\mathbb{Z}^o}) \\ &= D_\mathbb{C} \times ((L_a \setminus \{\mathrm{cr}\}) \to \mathbb{I}_{\mathbb{Z}^o}) \end{array}$$

where $\mathbb{I}_{\mathbb{Z}^o}$ denotes the set of intervals of $\mathbb{Z}^o$.

The definitions of the **guard** and **assign** operators for this domain are straightforward.

However this very simple choice for $D_0^\sharp$ does not allow to propagate and check properly the translated invariant of example 4 as shown in the next subsection.

We describe here the contribution to $F_a^\sharp$ of all the instructions in a program P_a, by defining the corresponding abstract transfer functions (in case $\phi_{l,l'}$ is not defined explicitly, it is equal to $\lambda\rho^\sharp \in D_0^\sharp.\bot$):

- "load integer" instruction $l : \texttt{li}\ \mathrm{r}_0,\ n; l' :\ \ldots$:

$$\phi_{l,l'}(\rho^\sharp) = \mathbf{assign}(\mathrm{r}_0, n, \rho^\sharp)$$

- "load" instruction $l : \texttt{load}\ \mathrm{r}_0,\ \underline{x}\,(v);\ l' : \ldots$:

$$\phi_{l,l'}(\rho^\sharp) = \mathbf{assign}(\mathrm{r}_0, M\{\underline{x} + v\}, \rho^\sharp)$$

- "store" instruction $l : \texttt{store}\ \mathrm{r}_0,\ \underline{x}\,(v);\ l' : \ldots$:

$$\phi_{l,l'}(\rho^\sharp) = \mathbf{assign}(M\{\underline{x} + v\}, \mathrm{r}_0, \rho^\sharp)$$

- "move register" instruction $l : \texttt{mr}\ \mathrm{r}_0,\ \mathrm{r}_1;\ l' : \ldots$:

$$\phi_{l,l'}(\rho^\sharp) = \mathbf{assign}(\mathrm{r}_0, \mathrm{r}_1, \rho^\sharp)$$

- "compare" instruction $l : \texttt{cmp}\ \mathrm{r}_0,\ \mathrm{r}_1;\ l' : \ldots$:

$$\phi_{l,l'}(\rho^\sharp) = \left\{ \begin{array}{l} \mathbf{assign}(\mathrm{cr}, \mathrm{LT}, \mathbf{guard}(\mathcal{T}, \mathrm{r}_0 < \mathrm{r}_1, \rho^\sharp)) \\ \sqcup\ \mathbf{assign}(\mathrm{cr}, \mathrm{EQ}, \mathbf{guard}(\mathcal{T}, \mathrm{r}_0 = \mathrm{r}_1, \rho^\sharp)) \\ \sqcup\ \mathbf{assign}(\mathrm{cr}, \mathrm{GT}, \mathbf{guard}(\mathcal{T}, \mathrm{r}_0 > \mathrm{r}_1, \rho^\sharp)) \end{array} \right.$$

- "conditional branching" instruction $l : \texttt{bc}(<)\ l''$; $l' : \ldots$:

$$\begin{array}{l} \phi_{l,l''}(\rho^\sharp) = \mathbf{guard}(\mathcal{T}, \mathrm{cr} = \mathrm{LT}, \rho^\sharp) \\ \phi_{l,l'}(\rho^\sharp) = \mathbf{guard}(\mathcal{F}, \mathrm{cr} = \mathrm{LT}, \rho^\sharp) \end{array}$$

  (the definition of the transfer functions for the conditional branching in case of other conditions is similar)
- "branching" instruction $l : \texttt{b}\ l''$; $l' : \ldots$:

$$\begin{array}{l} \phi_{l,l''}(\rho^\sharp) = \rho^\sharp \\ \phi_{l,l'}(\rho^\sharp) = \bot \end{array}$$

- "arithmetic" instruction $l : \texttt{op}\ \mathrm{r}_0,\ \mathrm{r}_1,\ \mathrm{r}_2;\ l' : \ldots$:

$$\phi_{l,l'}(\rho^\sharp) = \mathbf{assign}(\mathrm{r}_0, \mathrm{r}_1 \oplus \mathrm{r}_2, \rho^\sharp)$$

  where $\oplus$ corresponds to the binary operator associated to the arithmetic instruction `op`.

Fig. 7. Assembly abstract semantic function $F_a^\sharp$

6.2 Practical Problems of Checking

We envisage here the propagation and the checking of the translated invariant given in the example 4. More precisely, we consider the propagation of the local invariant corresponding to the program point l_2^a; we derive local invariants for the program points $l_3^a, l_4^a, l_5^a, l_6^a$ and l_{11}^a. The result is shown in figure 8 (the translated local invariants $I_a(l_6^a)$ and $I_a(l_{11}^a)$ associated to the program points l_6^a and l_{11}^a are recalled in the second part of the table).

No precise information about the value of the condition register cr is discovered after the comparison instruction: At l_5^a, cr is mapped to the non informative

Program point l	cr	$\underline{x}$	r_0	r_1
Propagated invariant starting from l_2^a				
l_2^a	$\top$	$[0, 100]$	$\top$	$\top$
l_3^a	$\top$	$[0, 100]$	$[0, 100]$	$\top$
l_4^a	$\top$	$[0, 100]$	$[0, 100]$	$[100, 100]$
l_5^a	$\top$	$[0, 100]$	$[0, 100]$	$[100, 100]$
l_6^a	$\top$	$[0, 100]$	$[0, 100]$	$[100, 100]$
l_{11}^a	$\top$	$[0, 100]$	$[0, 100]$	$[100, 100]$
Translated invariant				
l_6^a	$\top$	$[0, 99]$	$\top$	$\top$
l_{11}^a	$\top$	$[100, 100]$	$\top$	$\top$

Fig. 8. Invariant propagation

abstract value $\top$. Hence, no precise characterization of the values of the variables is inferred for any of the branches after the conditional branching instruction and the checking fails both at point l_6^a (since $[0, 100] \not\subseteq [0, 99]$) and at point l_{11}^a (since $[0, 100] \not\subseteq [100, 100]$).

The reason why no information about the value of the condition register is derived stems from the non-relational structure of the domain $D_0^\sharp$. Indeed, the choice made for $D_0^\sharp$ does not allow to take into account any relation between the value of cr and the values stored in the other memory locations (which is necessary for the invariant checking to succeed): In the above case, cr contains LT if $r_0 \in [0, 99]$; similarly, it contains EQ if $r_0 = 100$ and it cannot be equal to GT. The design of a new domain which solves this problem is addressed in section 6.3; roughly speaking, it is based on a partitioning of the abstract values by the value of the condition register.

A second issue is related to the fact that the comparison instruction compares the value contained in registers even if these registers stand for variables (in the example program of figure 3(c), r_0 contains the same value as the memory cell of address $\underline{x}$). The abstract transfer function for `cmp` given in figure 7 would not take into account this equality in case the abstract domain $D_0^\sharp$ is unable to carry some kind of equality relation between the values stored in distinct memory locations. Hence, a more precise domain is needed in order to fix this weakness of the initial domain $D_0^\sharp$. This second extension is described in section 6.4.

6.3 Value Partitioning

We suppose here that a domain $D_0^\sharp$ was defined for the assembly programs as in section 6.1 and we extend it to a new and more precise domain $D_1^\sharp$. An abstract value of $D_1^\sharp$ encloses an abstraction of the set of stores which map the condition register to c where c is any given condition register value. The set of stores which map cr to LT is approximated by an element of $D_0^\sharp$ (and the same for EQ and GT).

More formally, $D_1^\sharp$ is defined as a partitioning domain:

Definition 9 (Partitioning domain). Given the domain $D_0^\sharp$ and the Galois connection $(\mathbb{P}(S_a), \subseteq) \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} (D_0^\sharp, \sqsubseteq)$, the corresponding partitioning domain $(D_1^\sharp, \dot{\sqsubseteq})$ is defined as follows:

$$D_1^\sharp = \mathbb{C} \longrightarrow D_0^\sharp$$

Furthermore it defines a Galois connection

$$(\mathbb{P}(S_a), \subseteq) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (D_1^\sharp, \dot{\sqsubseteq})$$

where the concretization function is given by:

$$\forall \rho^\sharp \in D_1^\sharp,\quad \gamma_1(\rho^\sharp) = \begin{cases} \{\rho \in \gamma_0(\rho^\sharp(\mathrm{LT})) \mid \rho(\mathrm{cr}) = \mathrm{LT}\} \\ \cup \{\rho \in \gamma_0(\rho^\sharp(\mathrm{EQ})) \mid \rho(\mathrm{cr}) = \mathrm{EQ}\} \\ \cup \{\rho \in \gamma_0(\rho^\sharp(\mathrm{GT})) \mid \rho(\mathrm{cr}) = \mathrm{GT}\} \end{cases}$$

Proof of the Galois connection: Straightforward. □

Note that the notion of partitioning presented in definition 9 can be extended to other data-types: For instance the partitioning of the abstract values by the value of one or several boolean variable(s) can improve the precision of static analysis (this refinement is widely used in [BCC+03]).

The extension of the abstract operators is rather straightforward (we use the index "0" for the operators of $D_0^\sharp$ and the index "1" for the operators of $D_1^\sharp$):

- Assignment operator:
  If $\rho^\sharp \in D_1^\sharp$, then an assignment to the condition register is handled as follows:

$$\mathbf{assign}_1(\mathrm{cr}, \mathrm{EQ}, \rho^\sharp) = \begin{cases} \mathrm{LT} \mapsto \bot \\ \mathrm{EQ} \mapsto \mathbf{assign}_0(\mathrm{cr}, \mathrm{EQ}, \rho_0^\sharp) \\ \mathrm{GT} \mapsto \bot \end{cases}$$

  where $\rho_0^\sharp = \rho^\sharp(\mathrm{LT}) \sqcup \rho^\sharp(\mathrm{EQ}) \sqcup \rho^\sharp(\mathrm{GT})$.
  The assignment of other values to cr is similar.
  If l denotes an assembly l-value (which cannot evaluate to cr) and e any assembly expression, then:

$$\mathbf{assign}_1(l, e, \rho^\sharp) = \lambda c \in \mathbb{C}.\mathbf{assign}_0(l, e, \rho^\sharp(c))$$

- Guard operator:
  If $\rho^\sharp \in D_1^\sharp$, then:

$$\mathbf{guard}_1(\mathcal{T}, \mathrm{cr} = \mathrm{LT}, \rho^\sharp) = \begin{cases} \mathrm{LT} \mapsto \rho^\sharp(\mathrm{LT}) \\ \mathrm{EQ} \mapsto \bot \\ \mathrm{GT} \mapsto \bot \end{cases}$$

  The other conditions depending on cr are handled in a similar way.
  If the condition expression c does not depend on the condition register and if b is a boolean, then:

$$\mathbf{guard}_1(b, c, \rho^\sharp) = \lambda c \in \mathbb{C}.\mathbf{guard}_0(b, c, \rho^\sharp(c))$$

The comparison instruction l : `cmp` $\mathrm{r}_0, \mathrm{r}_1; l'$: ... is now analyzed as follows:

$$\phi_{l,l'}(\rho^\sharp) = \begin{cases} \mathrm{LT} \mapsto \mathbf{guard}_0(\mathcal{T}, \mathrm{r}_0 < \mathrm{r}_1, \rho_0^\sharp) \\ \mathrm{EQ} \mapsto \mathbf{guard}_0(\mathcal{T}, \mathrm{r}_0 = \mathrm{r}_1, \rho_0^\sharp) \\ \mathrm{GT} \mapsto \mathbf{guard}_0(\mathcal{T}, \mathrm{r}_0 > \mathrm{r}_1, \rho_0^\sharp) \end{cases}$$

where $\rho_0^\sharp = \rho^\sharp(\mathrm{LT}) \sqcup \rho^\sharp(\mathrm{EQ}) \sqcup \rho^\sharp(\mathrm{GT})$.

In practice, the partitioning can be implemented lazily. Indeed, the condition register is used only for tests; hence, its value is of interest only at some points of a program (between a comparison instruction and a conditional branching instruction, i.e. only at the program point l_5^a in our example). Lazy partitioning may allow memory savings: The real Power PC architecture features 8 condition register fields which makes lazy partitioning quite useful. Memory savings can also be achieved by using sharing.

Moreover, the partitioning layer (corresponding to $D_1^\sharp$) provides all the information we need about the condition register value and the relation between its value and the values of the other variables; hence, the basic domain $D_0^\sharp$ can be simplified into a domain which does not take the condition register into account (i.e. a function which maps integer registers and memory cells to intervals in the case of the domain chosen in section 6.1).

Example 5. Using the partitioning domain based on the interval domain yields the invariant displayed in figure 9. Note that we do lazy partitioning here: The mention $\forall c$ in the cr column means that the abstract store $\rho^\sharp$ depicted in the corresponding row maps any value of the condition register to the same element of $D_0^\sharp$ (no partitioning at this point). As remarked above, l_5^a is the only program point at which partitioning is absolutely necessary; hence, the values for all the partitions are merged after the branching (i.e. for the propagated invariants corresponding to the labels l_6^a and l_{11}^a).

The correct ranges for the register r_0 are now derived. However, the checking still fails since the ranges for the content of the memory cell $M\{\underline{x}\}$ do not take into account the test on r_0 (the value in r_0 is equal to the content of $M\{\underline{x}\}$). This issue motivates the next subsection.

Program point l	cr	$\underline{x}$	r_0	r_1
Propagated invariant starting from l_2^a				
l_2^a	$\forall c$	$[0, 100]$	$\top$	$\top$
l_3^a	$\forall c$	$[0, 100]$	$[0, 100]$	$\top$
l_4^a	$\forall c$	$[0, 100]$	$[0, 100]$	$[100, 100]$
l_5^a	LT	$[0, 100]$	$[0, 99]$	$[100, 100]$
	EQ	$[0, 100]$	$[100, 100]$	$[100, 100]$
	GT	$\bot$	$\bot$	$\bot$
l_6^a	$\forall c$	$[0, 100]$	$[0, 99]$	$[100, 100]$
l_{11}^a	$\forall c$	$[0, 100]$	$[100, 100]$	$[100, 100]$
Translated invariant				
l_6^a	$\forall c$	$[0, 99]$	$\top$	$\top$
l_{11}^a	$\forall c$	$[100, 100]$	$\top$	$\top$

Fig. 9. Invariant propagation with partitioning

6.4 Equalities Domain

As mentioned in the example 5, the abstract domain used for checking the invariant should keep information about equality relations between the content of distinct memory locations. In case the domain is not precise enough to express and derive such properties, we propose here to do a reduced product [CC79] with a specialized domain $D_e^\sharp$, which we define below:

Definition 10 (Variables equalities domain). The *equality domain* $(D_e^\sharp, \sqsubseteq_e)$ is defined by:

. $D_e^\sharp$ is the set of the *partitions* of the set of assembly memory locations V_a:

$$D_e^\sharp = \{(E_i)_{i\in I} \mid (\forall i \in I,\ E_i \subseteq V_a) \wedge (\cup_{i\in I} E_i = V_a) \\ \wedge (i \neq j \Rightarrow E_i \cap E_j = \emptyset) \\ \wedge (\forall i \in I,\ E_i \neq \emptyset)\}$$

. $\sqsubseteq_e$ is the inverse of the *sharpness* order:

$$(E_i)_{i\in I} \sqsubseteq_e (E_j)_{j\in J} \iff \forall j \in J,\ \exists i \in I,\ E_i \subseteq E_j$$

Moreover, this domain defines a Galois connection as follows:

$$(\mathbb{P}(S_a), \subseteq) \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} (D_e^\sharp, \sqsubseteq_e)$$

where:

$$\gamma_e((E_i)_{i\in I}) = \{\rho \in S_a \mid \forall i \in I,\ \exists v \in R_a, \\ \forall x \in E_i,\ \rho(x) = v\}$$

Proof of the Galois connection: Straightforward (the abstraction function is determined by the data of γ_e). □

Intuitively, the memory locations x and y may belong to the same element of the partition only if they store the same value.

Abstract operators **assign** and **guard** can be defined for the domain $D_e^\sharp$:

– Assignment operator:
The most important case is the "copy" assignment (the content of a memory location is copied into another one):

$$\mathbf{assign}(x, y, (E_i)_{i\in I}) = (E'_j)_{j\in J}$$

where the partition $(E'_j)_{j\in J}$ is defined completely by:

$$\begin{array}{ll} x \notin E_i \wedge y \notin E_i & \Longrightarrow \exists j \in J,\ E'_j = E_i \\ \{x\} \subset E_i \wedge y \notin E_i & \Longrightarrow \exists j \in J,\ E'_j = E_i \setminus \{x\} \\ x \in E_i \wedge y \in E_i & \Longrightarrow \exists j \in J,\ E'_j = E_i \\ x \notin E_i \wedge y \in E_i & \Longrightarrow \exists j \in J,\ E'_j = E_i \cup \{x\} \end{array}$$

The instructions `load`, `store` and `mr` fall in that case. We can remark that this case allows to derive new

information: Either x and y are equal before the assignment and this information is preserved or x and y are not equal before the assignment and then the equality $x = y$ is taken into account (after the other equalities involving x are relaxed).
The case of more complicated assignments is handled in a straightforward way. If e is a more complex expression,

$$\mathbf{assign}(x, e, (E_i)_{i\in I}) = (E'_j)_{j\in J}$$

where the partition $(E'_j)_{j\in J}$ is defined by:

$$\begin{array}{r} \exists j \in J,\ E'_j = \{x\} \\ x \notin E_i \implies \exists j \in J,\ E'_j = E_i \\ \{x\} \subset E_i \implies \exists j \in J,\ E'_j = E_i \setminus \{x\} \end{array}$$

This intuitively amounts to relaxing the equalities x was involved in before the assignment without deriving any new relation.

- Guard operator:
The guard operator does not allow to derive more information:

$$\mathbf{guard}(b, c, (E_i)_{i\in I}) = (E_i)_{i\in I}$$

Moreover the merge $(E_i)_{i\in I} \sqcup_e (E'_j)_{j\in J}$ of two partitions $(E_i)_{i\in I}$ and $(E'_j)_{j\in J}$ is the coarsest partition $(E''_k)_{k\in K}$ which is finer than both $(E_i)_{i\in I}$ and $(E'_j)_{j\in J}$:

$$\{E''_k \mid k \in K\} = \{E_i \cap E_j \mid i \in I \wedge j \in J\} \setminus \{\emptyset\}$$

The reduced product domain. We assume that the current assembly abstract domain $D^\sharp_1$ cannot deal with equalities between the content of memory locations (like non-relational domains and in particular like the interval domain considered above) and we strengthen it into a new domain $D^\sharp_2$ which can do it.

More precisely, we define $D^\sharp_2$ as a reduced product:

$$D^\sharp_2 = D^\sharp_1 \times D^\sharp_e$$

which defines the following Galois connection (with the product order):

$$(\mathbb{P}(S_a), \sqsubseteq) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (D^\sharp_2, \sqsubseteq)$$

Intuitively an element $(\rho^\sharp_1, (E_i)_{i\in I})$ represents a set of stores which are both upper-approximated by $\rho^\sharp_1$ and by $(E_i)_{i\in I}$:

$$\begin{array}{l} \forall (\rho^\sharp, (E_i)_{i\in I}) \in D^\sharp_2, \\ \quad \gamma_2(\rho^\sharp, (E_i)_{i\in I}) = \gamma_1(\rho^\sharp) \cap \gamma_e((E_i)_{i\in I}) \end{array}$$

A reduce operator $\mathbf{reduce} : D^\sharp_2 \longrightarrow D^\sharp_2$ is a function which transforms an abstract value into another one which has the same concretization (i.e. represents the same set of stores) by refining the first element: Taking equalities into account allows to derive more precise information in the domain $D^\sharp_1$ (more precise ranges can be found for some variables which turn out to be equal to other variables by intersecting their ranges). For instance in the case of the interval domain, a valid **reduce** operator would map $(\rho^\sharp, (E_i)_{i\in I})$ to $(\rho^\sharp_r, (E_i)_{i\in I})$ where the new abstract value $\rho^\sharp_r$ is defined by:

$$\forall x \in V_a,\ \text{if } x \in E_i,\ \text{then } \rho^\sharp_r(x) = \bigcap_{y\in E_i} \rho^\sharp(y)$$

In practice, the reduction operator can be integrated to the assign and the guard operators.

Program point l	cr	$\underline{x}$	r_0	r_1
Propagated invariant starting from l^a_2				
l^a_2	$\forall c$	$[0, 100]$	$\top$	$\top$
l^a_3	$\forall c$	$[0, 100]$	$[0, 100]$	$\top$
l^a_4	$\forall c$	$[0, 100]$	$[0, 100]$	$[100, 100]$
l^a_5	LT	$[0, 99]$	$[0, 99]$	$[100, 100]$
	EQ	$[100, 100]$	$[100, 100]$	$[100, 100]$
	GT	$\bot$	$\bot$	$\bot$
l^a_6	$\forall c$	$[0, 99]$	$[0, 99]$	$[100, 100]$
l^a_{11}	$\forall c$	$[100, 100]$	$[100, 100]$	$[100, 100]$
Translated invariant				
l^a_6	$\forall c$	$[0, 99]$	$\top$	$\top$
l^a_{11}	$\forall c$	$[100, 100]$	$\top$	$\top$

Fig. 10. Invariant checking with partitioning and equalities

Example 6 (Equalities). In the example program of figure 3(c), the equalities domain discovers the equality $M\{\underline{x}\} = \mathrm{r}_0$ at points $l^a_3, l^a_4, l^a_5, l^a_6$ and l^a_{11} (we only consider here the program points we need to consider in order to propagate and check the local invariant of the point l^a_2 as done in example 4 and in example 5).

The reduction improves the ranges for the content of the memory cell of address $\underline{x}$ at point l^a_5: In case cr is set to LT, then the content of r_0 is in the range $[0, 99]$; hence, so is the content of variable x.

The resulting local invariants given in figure 10 allow the checking to succeed: Indeed, the local invariant computed for point l^a_6 starting from the translated invariant of point l^a_2 is more precise than the translated local invariant for point l^a_6 (and the same for l^a_{11}); hence, the checking condition given in section 5.2 is satisfied.

7 Implementation and Results

This section presents an overview of the implementation of a prototype of an assembly code certifier and assesses the results of this experience.

7.1 Context

The purpose here is to design a prototype able to certify assembly programs corresponding to typical embedded

systems, like those considered in [BCC+02, BCC+03]. The certification of a large class of C programs (i.e. automatic analysis resulting in very low false alarms number) is not our current goal; hence, we restricted to a class of more simple, yet safety critical C programs.

These programs are written in C but mainly use rather basic features. The control structure of these programs involves procedures (i.e. void functions) and a few more complicated functions (with complex arguments and a return value). The data-types which should be handled do not include pointers even if pointers are implicitly used when passing arrays to functions (the arguments passed by reference can always be determined without any ambiguity, so an alias analysis was unnecessary). Most classical C data-types are widely used: Various integer and floating point data-types, structures, arrays and enums data-types. A pleasant aspect of the class of programs under consideration is that they do not use recursion. Therefore, the calling stack (the sequence of function calls) can be represented explicitly during the analysis. The absence of dynamic memory allocation and of recursion also implies that the set of memory locations (in the current environment and in the calling functions) can be represented explicitly and finitely at any program point, which simplifies the analysis and makes it more precise.

The target architecture we chose comprises a 64 bytes version of the motorola PowerPC processor [Mot97] and a version of **gcc** (we used a cross-compiler). The assembly language introduced in section 2.5 is a simplified version of the PowerPC instruction set; however, the real architecture is much more complicated. Indeed, the processor we considered features 32 "General Purpose Registers" (integer registers), 32 "Floating-Point Registers" and a "Condition Register" composed of 8 fields. Memory access proceeds through various addressing modes; the relative addressing described in section 2.5 is a generalization of the main addressing modes.

The compilation of programs containing functions and procedures involves an execution stack. Local variables are addressed relatively to the stack pointer (function parameters are also stored in the stack). Therefore, the precise analysis of the structure of the stack is crucial for the checking to succeed.

7.2 Structure of the Prototype

Structure. The source analyzer is quite similar to the C analyzers described in [BCC+02, BCC+03]; however it does not include all the domain refinements considered there. We provide more details about the abstract domain we used below. The source analyzer checks the correctness of the source code (as sketched in section 4.1).

The invariant translator preprocesses STABS standard debugging information and inputs the invariant produced by the source analyzer. The result of the invariant translation corresponds to the invariant denoted by J_a in section 5.2.

The assembly invariant checker proceeds to the propagation and to the verification of the translated invariant as described in section 5.2. The resulting invariant can be dumped to the disk as a bunch of html files, which allows to inspect manually the final results of the analysis (additional information about the translation are also output as html files).

The assembly checker also carries out the assembly code certification. This involves the checking of the following properties:

- The arithmetic instruction do not yield any exception (no division by 0 or overflow error may occur);
- The access to memory is safe: Any load or store instruction only affect defined and authorized memory locations (i.e. no segmentation fault may occur)

In fact the treatment of arithmetic exception may be modified by the user. Therefore we plan to make the precise nature of the errors the analyzer should keep track a parameter of the analysis.

The whole development (frontends, analyzers, translator and checker) amounts to 25 000 lines of OCaml code and required three months of full-time work for one person.

Abstract domain. The abstract domain is more complicated than the interval domain considered along the paper; nevertheless the content of sections 4, 5 and 6 can be straightforwardly generalized. Basic integer and floating point objects are abstracted to intervals. A boolean type defined as an enum type is precisely handled (using a domain of constants). The abstract domain represents exactly the structure of composed objects (arrays, structures and enums); the basic members of this structures (integer of floating point array cells and structure members) are abstracted in the same way as simple variables (using intervals). Moreover the abstract domain presents the ability of partitioning stores using control-based criteria (similarly to the approach of [HT98]). A parameter of the analyzer commands the control-based partitioning by pointing out which control structure (conditional or loop) should be analyzed precisely.

At the assembly level, the domain is quite similar but the refinements described in section 6 (lazy partitioning by the value of the condition register and reduced product with the equalities domain) are required and apply straightforwardly. Moreover the importance of pointers at the assembly level (for representing arrays, structures and the stack pointer) makes their precise abstract representation crucial. The abstract representation of a pointer $\underline{x}$ is a function $\phi_{\underline{x}}$: $\phi_{\underline{x}}$ maps an integer n to the cell of the abstract domain which corresponds to the concrete memory location of address $\underline{x} + n$ (intuitively $\phi_{\underline{x}}$ inputs an offset and outputs the abstract representation

of the corresponding cell). In case $\underline{x}$ corresponds to an array, $\phi_{\underline{x}}$ maps the valid indexes for this array to the abstract value corresponding to its cells. This symbolic representation renders the checking of the correctness of memory access simple: `load` $r_0, \underline{x}(r_1)$ is correct if and only if the register r_1 contains a value which defines a correct offset for the pointer corresponding to $\underline{x}$.

Remark 3 (Memory alignments). In fact, the problem of memory alignments required the implementation of an additional domain. The assembly language introduced in section 2.5 features one basic data-type only and ignores the problem of memory alignments: All the memory cells have size 1, so the addresses of the cells of an array of integers are successive integers. In the case of the real PowerPC processor, integers and floating point numbers are 4 bytes long whereas short integers are 2 bytes long. In case of an integer array lookup, the interval information is generally not sufficient in order to prove the correctness of the memory access. For instance, if we consider an array of floating point numbers, the addresses of the cells are multiples of 4 and if the index in the source program is in the range $[a, b]$, then the assembly offset is in the range $[4a, 4b]$; if $a < b$, then $4a + 1$ belongs to the interval but does not correspond to a valid address since the addresses are multiples of 4. The congruence domain [Gra89] provides adequate information to prove the correctness of arrays and struct reads and writes; so a reduced product with this domain can be defined as in section 6.4.

The abstract operators have been extended to convey congruence information in the prototype.

Example 7. We give here a few details about an example run of the prototype on a C program of 400 lines, containing 10 functions and about 50 global variables. One of the loops of the program required a precise analysis (i.e. partitioning of traces by the number of iterations in the loop). A main loop controls the execution of almost all the program (the number of iterations in this loop is unbounded). A few unrolling iterations (the union operator is used for the first iterations) and the use of a staged widening with threshold [BCC+02] were necessary for the source analyzer to produce a quite precise invariant. A list of values for the widening threshold and the program points at which control partitioning should be done are parameters of the analyzer.

The source analysis requires 2.5 s. and 15 Mb of RAM on an Intel Pentium III laptop (1 GHz) under linux 2.4.18. It produced one false alarm, which would be solved using a more precise abstract domain (section 6.7 of [BCC+02]).

The parsing of the assembly program including the processing of the debugging information and the building of the mappings π_l and π_v requires about 4.5 s. The invariant translation requires 1.5 s.

The invariant propagation is done in 4.1 s; the checking of the stability of the translated invariant is passed after about 1 s (it actually succeeds). Checking requires about 27 Mb of RAM. The final assembly analysis leaves the same false alarm as for the source (one potential overflow).

The prototype succeeded in proving the soundness of invariants. However, the size of the programs we could consider is fairly limited: The prototype was not designed with the purpose of scaling-up, since it was a first experience of invariant translator. The main limitation comes from the memory usage and stems from the fact the assembly invariant was completely generated. A real tool would generate and check it incrementally (the memory usage would not be much greater than that of the source analyzer).

8 Conclusion

We proposed a method for certifying assembly programs produced by compilation from programs written in a source language we have an analyzer for. The method is generic with respect to the compiler and to the choice of an abstract domain for representing sets of stores (since the assembly abstract domain is derived from the source abstract domain). Invariant propagation and checking may require a precise treatment of some assembly language aspects; nevertheless, we have to cope with this additional issues only once even if the compiler is modified or changed, since it merely stems from the characteristics of the assembly language itself.

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 the final result apart from any hypothesis on the correctness of the rest of the process is always 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 current prototype into a true certifying tool, by extending the abstract domain to a relational domain and the source language under consideration. Another more challenging goal would be to define a class of transformations (optimizations...) the method would work for and to augment this class by taking more optimizations into account. This would certainly require the extension of the definition of the correctness of compilation. A last direction would be to use similar methods to analyze programs generated automatically from a specification: A specification could be used to compute an invariant on the program (the specification should contain appropriate information about the program behavior); checking the invariant on the program being simpler than inferring an invariant from the generated program alone. Analyzing a rather "high-level" specification may make the

inference of properties more simple and thus increase the precision of static analysis.

Acknowledgments. We deeply thank the anonymous referees for their significative comments on an early version of this paper. We also would like to thank Bruno Blanchet, Patrick and Radhia Cousot, Jérôme Feret, Charles Hymans, Laurent Mauborgne, Antoine Miné, and David Monniaux for stimulating discussions.

References

[AFMW96] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Static Analysis Symposium (SAS'96)*, volume 1145 of *LNCS*, pages 51–66, September 1996.

[App01] A. W. Appel. Foundational Proof-Carrying Code. In *Proceedings of the 16th Symposium on Logics in Computer Science (LICS'01)*, pages 247–256, Boston (USA), june 2001.

[BCC+02] 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 *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, 2002.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Languages, Design and Implementation (PLDI'03)*, pages 196–207, San Diego (USA), june 2003.

[Ber98] Y. Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, 1998.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735:128–141, 1993.

[CC77] 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 (POPL'77)*, pages 238–252, Los Angeles (California, USA), January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, San Antonio, Texas, January 1979. ACM Press, New York, NY.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC02] 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 (POPL'02)*, pages 178–190, Portland, Oregon, January 2002. ACM Press, New York, NY.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97, Tucson, Arizona, January 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Cou97] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[Gra89] P. Granger. Static Analysis of Arithmetical Congruences. *Int. J. Computer. Math.*, 30:165–190, 1989.

[HT98] M. Handjieva and S. Tzolovski. Refining Static Analyses by Trace-Based Partitioning Using Control Flow. In *Proceedings of the 5th Static Analysis Symposium (SAS'98)*, LNCS, pages 200–214, Pisa (Italy), september 1998.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[MCG+99] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, and D. Walker. TALx86: A Realistic Typed Assembly Language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, may 1999.

[Min01] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs As Data Objects (PADO II)*, volume 2053 of *LNCS*, Aahrus (Denmark), 2001.

[Mot97] Motorola. *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, 1997.

[MTC+96] 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*, Tucson, AZ, February 1996.

[Nec97] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, 1997.

[Nec00] G. C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*

(PLDI'00), pages 83–94, Vancouver, Canada, June 2000.

[NL98] G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Languages, Design and Implementation (PLDI'98)*, pages 333–344, Montréal, Canada, june 1998.

[PSS98] A. Pnueli, O. Shtrichman, and M. Siegel. Translation Validation for Synchronous Languages. In *Prooceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, pages 235–246, Aahrus, Denmark, July 1998.

[Riv03] X. Rival. Abstract Interpretation-based Certification of Assembly Code. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, pages 41–55, New York (USA), january 2003.

[Str02] M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77, Copenhagen, Denmark, July 2002. Springer Verlag.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TF98] 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*, pages 144–153, Madrid, Spain, dec 1998.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Seperate Cache and Path Analyses. *Real-Time Systems*, 18:157–179, 2000.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[ZPFG02] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A Translation Validator for Optimizing Compilers. In *Electronic Notes in Theoretical Computer Science*, volume 65, 2002.