Symbolic Transfer Function-based Approaches to Certified Compilation *

Xavier RIVAL
Département d'Informatique,
École normale supérieure,
45, rue d'Ulm, 75230, Paris Cedex 5, France
rival@di.ens.fr

Abstract

We present a framework for the certification of compilation and of compiled programs. Our approach uses a symbolic transfer functions-based representation of programs, so as to check that source and compiled programs present similar behaviors. This checking can be done either for a concrete semantic interpretation (Translation Validation) or for an abstract semantic interpretation (Invariant Translation) of the symbolic transfer functions. We propose to design a checking procedure at the concrete level in order to validate both the transformation and the translation of abstract invariants. The use of symbolic transfer functions makes possible a better treatment of compiler optimizations and is adapted to the checking of precise invariants at the assembly level. The approach proved successful in the implementation point of view, since it rendered the translation of very precise invariants on very large assembly programs feasible.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Formal methods, Validation, Assertion checkers, Correctness proofs; D.3.1 [Formal Definitions and Theory]: Semantics; D.3.4 [Processors]: Compilers, Optimization; F.3.1 [Specifying and Verifying and Reasoning about Programs]: Invariants, Mechanical verification; F.3.2 [Semantics of Programming Languages]: Operational semantics, Denotational semantics, Algebraic approaches to semantics, Program analysis.

General Terms: Algorithms, Design, Theory, Verification

Keywords: Static Analysis, Abstract Interpretation, Certification, Compilation, Translation Validation

1 Introduction

1.1 Motivations

Critical software (as in embedded systems) is concerned both with correctness and safety. The designer of such systems is usually interested in checking that the final program correctly implements a specification and is safe in the sense that it should not yield any "bad behavior". Hence, much work has been done for the analysis of source programs [4, 5]. However, in the same time compilers have become huge and complex. For instance, the code of **gcc** amounts to about 500 000 lines of code. Moreover, the semantics of the C language leaves some erroneous cases unspecified in order to let the compiler designer free to implement various optimizations. Therefore, certifying compilation and compiled programs turns out to be necessary in the case of highly critical applications. Given a source program P_s and a compiled program P_c, the following questions are of interest: (1) does P_c yield the same behaviors as P_s ? (2) does P_c meet some safety requirements; e.g. is it runtime error free ?

Proving the compiler formally would be an answer for the first question, but this solution is usually too expensive and is not practical if the code of the compiler is not available. Hence, a more practical approach proceeds by proving the semantic equivalence of P_s and P_c (Translation Validation, noted TV). In practice, an external tool generates proof obligations and attempts to check them after each compilation. Analyzing directly assembly programs is problematic since compilation induces a loss of structure both at the data and at the control level. Therefore, a more satisfactory approach to the second problem is based on the analysis of the source program P_s and on a phase of Invariant Translation (noted IT), which yields some sound invariant for P_c.

A unifying framework for the certification of compilation and of compiled programs would allow to compare these methods and to share the effort to tackle some common problems: finding suitable representations for programs, characterizing the semantic relation between source and compiled programs, handling optimizations...

1.2 Related Work

In Proof Carrying Code systems [16, 18, 2], an untrusted compiler is required to provide annotations together with the compiled program, which is related to IT. Before the program is executed, a certifier generates Verification Conditions and attempts to prove them using the annotations; if it succeeds, then the program obeys the safety policy and can be executed. Typed Assembly and Intermediate Languages (TAL/TIL) were proposed as a means to keep more semantic information in the target programs [15, 21], so as to improve the level of confidence into compiled programs ([14] relates the implementation of a compiler for a safe C dialect). The TAL approach was further extended by [22] with more expressive annotations. Yet, all these systems require deep compiler modifications. Moreover, most of them aim at checking type safety properties and are restricted to security or memory safety properties.

As an example of abstract invariant translation system, we can cite our previous work in [20]: a source analyzer was instrumented to translate the source invariant into an invariant which was then proved to hold for the compiled program. The process does not require the compiler to be modified. Complex invariants were translated and checked at the assembly level so as to prove the absence

*This work was supported by the ASTREE Project of the French RNTL

POPL'04, January 14–16, 2004, Venice, Italy.

of runtime errors. Yet, these results can be improved, so as to handle more complex and precise invariants, to scale up and to accommodate optimizations: indeed, our tool was not able to translate relational invariants or to certify optimized code.

Translation Validation-based methods were pioneered in [19] and extended in [17, 23]. An external tool generates Verification Conditions and attempts to solve them, so as to prove the compilation correct. The compiler does not need to be modified.

As far as we know, the feasibility of an IT system based on TV has not been remarked before.

1.3 Contribution

Our purpose is to provide a framework for the certification of compilation and of compiled programs and to propose an efficient, flexible and scalable IT process:

- We provide a symbolic representation of (source and assembly) programs so as to make compilation certification suitable; then, we formalize TV- and IT-based approaches in this framework.
- A new approach to Invariant Translation based on Translation Validation appears in this framework. Indeed, the equivalence checking step justifies the semantic correctness of compilation; hence, it discards the abstract invariant checking requirement of [20]. Furthermore, this new approach proves the semantic equivalence of source and compiled programs and takes benefit from the TV pass to improve the efficiency of the invariant translation.
- We implemented the latter approach. The certifier was run on real examples of highly critical embedded software, for the purpose of proving the safety of assembly programs using the same criteria as for source programs in [5], which requires to translate very precise invariants. We are not aware of any similar existing work.
- At last, the symbolic representation of programs turns out to be a very adequate model to tackle the difficulties which arise when considering optimizations.

1.4 Overview

Sect. 2 introduces some preliminaries and states the definition of compilation correctness used in the paper. Sect. 3 sets up a symbolic representation of programs adequate for compilation certification (either by TV or IT). IT and TV are integrated to this framework in Sect. 4, together with a new approach to Invariant Translation, based on Translation Validation. Experimental results are provided and show the scalability of the latter approach. Sect. 5 tackles the problem of optimizing compilation using the technical tools introduced in Sect. 3. Sect. 6 concludes.

2 Correctness of Compilation

This section introduces preliminaries and leads to a definition of compilation correctness in the non-optimizing case.

2.1 Programs, Transition Systems, Symbolic Transfer Functions

Programs are usually formalized as *Labeled Transition Systems* (LTS). An execution state is defined in this model by a program point l (a *label*) and a *store* ρ, that is a function which assigns a value to each memory location (variable, register, array cell). In the following, we write X for the set of memory locations, L for the set of labels, R for the set of values and $S = X \to R$ for the set of stores. In this setting, a program P is a tuple $(L_P, \to_P, i_P)$ where L_P is the set of labels of P, $i_P \in L_P$ is the entry point of P and $(\to_P) \subseteq (L_P \times S) \times (L_P \times S)$ is the transfer relation of P. Intuitively, $(l_0, \rho_0) \to_P (l_1, \rho_1)$ if and only if an execution of P reaching point l_0 with store ρ_0 may continue at program point l_1, with store ρ_1. This model allows non-determinism since $(\to_P)$ is a relation. Note that the notion of program point does not necessarily correspond to the notion of syntactic program point. A label may in particular define both a calling context and a program point.

The semantics of a program P is the set of all the possible runs of P. In the following, if $\mathcal{E}$ is a set, we note $\mathcal{E}^\star$ for the set of the finite sequences of elements of $\mathcal{E}$ and $\mathbb{P}(\mathcal{E})$ for the power-set of $\mathcal{E}$. An *execution trace* (or run) is a sequence of states. The *semantics* $[\![P]\!]$ of a program $P = (L_P, \to_P, i_P)$ is the set of the finite partial execution traces of P: $[\![P]\!] \subseteq (L_P \times S)^\star$. The semantics $[\![P]\!]$ boils down to a least fixpoint (lfp) in the complete lattice $(\mathbb{P}((L_P \times S)^\star), \subseteq)$: $[\![P]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset} F_P$ where $F_P : \mathbb{P}((L_P \times S)^\star) \to \mathbb{P}((L_P \times S)^\star)$ is the continuous semantic function defined below:

$$F_P(\mathcal{X}) = \{\langle (x_0,s_0),\ldots,(x_n,s_n),(x_{n+1},s_{n+1})\rangle \mid \langle (x_0,s_0),\ldots,(x_n,s_n)\rangle \in \mathcal{X} \wedge (x_n,s_n) \to_P (x_{n+1},s_{n+1})\} \cup \{\langle (i_P,s)\rangle \mid s \in S\}$$

Source and Assembly languages: In the following, we choose a subset of C as a source language (Fig. 1(a)) and the Power-PC-like assembly language presented in Fig. 1(b). The source language features loops, conditionals, assignments, array and integer variables. Source statements and assembly instructions are (implicitly) labeled.

The assembly language provides registers (denoted by r_i), memory, indirections in the data and usual assembly features. The address of a variable x is denoted by $\underline{x}$. If n is an integer, the predicate **isaddr**(n) means that n is a valid address for a memory location in the current program; then, **content**(n) denotes the content of the memory cell of address n. If **isaddr**$(\underline{x}+v)$ holds, then the instruction `load` $r_0, \underline{x}(v)$ loads the value **content**$(\underline{x}+v)$ into register r_0; in case $\underline{x}+v$ is not a valid address, the load instruction yields an error. The `store` instruction is similar (but copies the content of a register into a memory cell). The instruction `li` r_0, n loads the integer n into the register r_0. The arithmetic instruction `op` r_0, r_1, r_2 takes the content of r_1 and r_2 as arguments and stores the result into r_0. The comparison instruction `cmp` r_0, r_1 sets the value of the condition register according to the order of the values in r_0 and r_1: if $r_0 < r_1$, cr is assigned LT and so on. The conditional branching instruction `bc`$(<)$ l directs the control to the instruction of label l if cr contains the value LT. The branching instruction `b` l re-directs the control to the instruction of label l.

The semantics of source and assembly programs is defined as above since they are transitions systems. Adding procedures, non-determinism, and more complex data structures would be straightforward.

2.2 Definition of Correct Compilation

Let $P_s = (L_s, \to_s, i_s)$ be a source program and $P_a = (L_a, \to_a, i_a)$ be an assembly program. We assume that P_a is a compiled program corresponding to P_s, so we expect P_s and P_a to present similar behaviors: execution traces of both programs should be in correspondence. The relation between traces of P_s and of P_a can be defined by a mapping of source and assembly program points and memory locations. For instance, on Fig. 2, the source program points $l^s_0, l^s_1, l^s_2, l^s_3, l^s_4, l^s_5, l^s_6$ respectively correspond to the assembly program points $l^a_0, l^a_2, l^a_4, l^a_8, l^a_{11}, l^a_{15}, l^a_{16}$. Similarly, the source variable t corresponds to a region of the assembly memory starting at address $\underline{t}$ and of the same size as t (we suppose memory cells have size 1) and so on. Then, the value of i at point l^s_4 should be equal to the value stored at address $\underline{i}$ at point l^a_8. These map-

$$
\begin{array}{rll}
\mathtt{Lv} ::= & x\,(x \in X) \mid \mathtt{Lv}[\mathtt{E}] & c \in \{<, \leq, =, \neq, >, \geq\} \\
\mathtt{E} ::= & n\,(n \in \mathbb{Z}) \mid \mathtt{Lv} & v \in \{r_0, \ldots, r_N\} \cup \mathbb{Z} \\
\mid & \mathtt{E} + \mathtt{E} \mid \mathtt{E} - \mathtt{E} & \mathtt{op} ::= \quad \mathtt{add} \mid \mathtt{sub} \\
\mid & \mathtt{E} \star \mathtt{E} \mid \mathtt{E} / \mathtt{E} & \mid \quad \mathtt{mul} \mid \mathtt{div} \\
\mathtt{C} ::= & \textbf{true} \mid \textbf{false} \mid \neg \mathtt{C} & n \in \quad \mathbb{Z} \\
\mid & \mathtt{C} \wedge \mathtt{C} \mid \mathtt{C} \vee \mathtt{C} & \mathtt{I} ::= \quad \mathtt{load}\, r_i, n(v) \\
\mid & \mathtt{E} == \mathtt{E} \mid \mathtt{E} < \mathtt{E} & \mid \quad \mathtt{store}\, r_i, n(v) \\
\mathtt{S} ::= & \mathtt{Lv} := \mathtt{E} & \mid \quad \mathtt{li}\, r_i, n \\
\mid & \textbf{if}(\mathtt{C})\,\mathtt{B}\,\textbf{else}\,\mathtt{B} & \mid \quad \mathtt{op}\, r_i, r_j, v \\
\mid & \textbf{while}(\mathtt{C})\,\mathtt{B} & \mid \quad \mathtt{cmp}\, r_i, r_j \\
\mathtt{B} ::= & \{\mathtt{S}; \ldots; \mathtt{S}\} & \mid \quad \mathtt{bc}(c)\, l \mid \mathtt{b}\, l
\end{array}
$$

(a) Source language (b) Assembly language

Figure 1. Syntax

i, x: integer variables
t: integer array of length $n \in \mathbb{N}$, where n is a parameter

l_0^s $i := -1;$
l_1^s $x := 0;$
l_2^s **while**$(i < n)\{$
l_3^s $\quad i := i + 1;$
l_4^s $\quad x := x + t[i]$
l_5^s $\}$
l_6^s $\ldots$

(a) Source program

l_0^a $\mathtt{li}\, r_0, -1$
l_1^a $\mathtt{store}\, r_0, \underline{i}(0)$
l_2^a $\mathtt{li}\, r_1, 0$
l_3^a $\mathtt{store}\, r_1, \underline{x}(0)$
l_4^a $\mathtt{load}\, r_0, \underline{i}(0)$
l_5^a $\mathtt{li}\, r_1, n$
l_6^a $\mathtt{cmp}\, r_0, r_1$
l_7^a $\mathtt{bc}(\geq)\, l_{16}^a$
l_8^a $\mathtt{load}\, r_0, \underline{i}(0)$
l_9^a $\mathtt{add}\, r_0, r_0, 1$
l_{10}^a $\mathtt{store}\, r_0, \underline{i}(0)$
l_{11}^a $\mathtt{load}\, r_1, \underline{x}(0)$
l_{12}^a $\mathtt{load}\, r_2, \underline{t}(r_0)$
l_{13}^a $\mathtt{add}\, r_1, r_1, r_2$
l_{14}^a $\mathtt{store}\, r_1, \underline{x}(0)$
l_{15}^a $\mathtt{b}\, l_4^a$
l_{16}^a $\ldots$

(b) Assembly program

Figure 2. Compilation

pings are further represented by a bijection $\pi_L : L_s^r \to L_a^r$ where $L_s^r \subseteq L_s$ and $L_a^r \subseteq L_a$ (the *program point mapping*) and a bijection $\pi_X : X_s^r \to X_a^r$ where $X_s^r \subseteq X_s$ and $X_a^r \subseteq X_a$ (the *variable mapping*). Note that $L_a^r \subset L_a$ in general, since intermediate assembly program points have no source counterpart as is the case for $l_1^a, l_3^a, l_5^a, \ldots$ In case some non reachable source program points are deleted, we also have $L_s^r \subset L_s$. The semantic correspondence between P_s and P_a can be stated as a one to one relation between the observational abstraction of the traces of both programs. Most optimizations do not fit in this simple framework; some are dealt with in Sect. 5. *Erasure operators* $\Phi_i : (L_i \times S)^\star \to (L_i^r \times S)^\star$ (for $i \in \{s, a\}$) can be defined, that abstract away program points and memory locations which do not belong to L_i^r and to X_i^r: $\Phi_i(\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle)$ is the trace $\langle (l_{k_0}, \rho'_{k_0}), \ldots, (l_{k_m}, \rho'_{k_m}) \rangle$ where $k_0 < \ldots < k_m$, $\{k_0, \ldots, k_m\} = \{j \mid l_j \in L_i^r\}$ and $\forall i$, ρ'_{k_i} is the restriction of ρ_{k_i} to X_i^r. The *observational abstraction operators* $\alpha_i^r : \mathbb{P}((L_i \times S)^\star) \to \mathbb{P}((L_i^r \times S)^\star)$ can be defined by $\alpha_i^r(\mathcal{X}) = \{\Phi_i(\sigma) \mid \sigma \in \mathcal{X}\}$ for $i \in \{s, a\}$. The stores ρ_s and ρ_a are equivalent if and only if $\forall x_s \in X_s^r$, $\forall x_a \in X_a^r$, $\pi_X(x_s) = x_a \Rightarrow \rho_s(x_s) = \rho_a(x_a)$ (we note $\rho_s \simeq_{\pi_X} \rho_a$ for the store equivalence). The traces $\sigma_s = \langle (l_0^s, \rho_0^s), \ldots, (l_n^s, \rho_n^s) \rangle$ and $\sigma_s = \langle (l_0^a, \rho_0^a), \ldots, (l_n^a, \rho_n^a) \rangle$ are equivalent (noted $\sigma_s \simeq \sigma_a$) if and only if $\forall i$, $l_i^a = \pi_L(l_i^s) \wedge \rho_i^s \simeq_{\pi_X} \rho_i^a$. If $\forall j \in \{s, a\}$, $T_j \subseteq \mathbb{P}((L_j^r \times S)^\star)$, we write $T_s \simeq T_a$ if and only if $\simeq$ is a bijection between T_s and T_a.

DEFINITION 1 (COMPILATION CORRECTNESS, [20]). *The compilation of P_s into P_a is correct with respect to π_L and π_X if and only if $\alpha_s^r([\![P_s]\!]) \simeq \alpha_a^r([\![P_a]\!])$ (i.e. iff $\alpha_s^r([\![P_s]\!])$ and $\alpha_a^r([\![P_a]\!])$ are in bijection).*

3 A Symbolic Representation of Programs

An atomic source statement is usually expanded at compile time into a long and intricate sequence of assembly instructions (a source statement can be compiled into many different sequences). Optimizations (Sect. 5) make the correspondence between source and assembly actions even more involved. Therefore, we design in this section a "higher level" symbolic representation of the semantics of programs which proves efficient to relate the behaviors of source and compiled programs.

3.1 Symbolic Transfer Functions

Symbolic Transfer Functions (STFs) appeared in the work of Colby and Lee [7] as a means to avoid accumulated imprecisions along paths in the control flow: an STF describes precisely the store transformation between two program points. STFs turn out to be also a nice setting for reasoning about program equivalence.

An STF δ is either a parallel assignment $\lfloor \overrightarrow{x} \leftarrow \overrightarrow{e} \rfloor$, where $\overrightarrow{x}$ and $\overrightarrow{e}$ are respectively a sequence of memory locations and a sequence of expressions; or a conditional construct $\lfloor c\,?\,\delta_t \mid \delta_f \rfloor$ where c is a condition and δ_t, δ_f are STFs; or the void STF $\square$ (which stands for the absence of transition). Note that the empty assignment is the identity; it is denoted by ι. We write $\mathbb{T}$ for the set of STFs.

Semantics of Symbolic Transfer Functions: An STF is interpreted by a function which maps a store ρ to the set of possible transformed stores in presence of non-determinism. In the following, we forbid non-determinism for the sake of concision; hence, a transfer function is interpreted by a function which maps a store to a store or to the constant $\bot_S$ (in case of error, there is no transformed store). We assume that the semantics of expressions and conditions is defined as follows: if e is an expression, then $[\![e]\!] \in S \to R$ and if c is a condition, then $[\![c]\!] \in S \to \mathbb{B}$ where $\mathbb{B}$ denotes the set of booleans $\{\mathcal{T}, \mathcal{F}\}$. The update of the variable x with the value v in the store ρ is denoted by $\rho[x \leftarrow v]$. The semantics of STFs is defined by induction on the syntax as follows:

- $[\![\square]\!](\rho) = \bot_S$
- $[\![\lfloor x_0 \leftarrow e_0, \ldots, x_n \leftarrow e_n \rfloor]\!](\rho) = \rho[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n]$, where $\forall i, [\![e_i]\!](\rho) = v_i$. Note that the values are all computed in the initial environment ρ.
- $[\![\lfloor c\,?\,\delta_t \mid \delta_f \rfloor]\!] = \begin{cases} [\![\delta_t]\!](\rho) & \text{if } [\![c]\!](\rho) = \mathcal{T} \\ [\![\delta_f]\!](\rho) & \text{if } [\![c]\!](\rho) = \mathcal{F} \end{cases}$

Composition of Symbolic Transfer Functions: The composition of two STFs can be written as an STF:

THEOREM 1 (COMPOSITION, [7]). *There exists a (non unique) computable operator $\oplus : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ such that $\forall \delta_0, \delta_1 \in \mathbb{T}$, $[\![\delta_0 \oplus \delta_1]\!] = [\![\delta_0]\!] \circ [\![\delta_1]\!]$.*

EXAMPLE 1. *If x and y are not aliased, if $\delta_0 = \lfloor y > 3\,?\,\lfloor z \leftarrow y + x \rfloor \mid \lfloor z \leftarrow 3 \rfloor \rfloor$ and $\delta_1 = \lfloor x < 4\,?\,\lfloor y \leftarrow x \rfloor \mid \square \rfloor$, then $\lfloor x < 4\,?\,\lfloor x > 3\,?\,\lfloor z \leftarrow 2x, y \leftarrow x \rfloor \mid \lfloor z \leftarrow 3, y \leftarrow x \rfloor \rfloor \mid \square \rfloor$ is a correct definition for $\delta_0 \oplus \delta_1$.*

The composition is generally not associative; yet, we do not distinguish $\delta_0 \oplus (\delta_1 \oplus \delta_2)$ and $(\delta_0 \oplus \delta_1) \oplus \delta_2$, since they have the same semantics. We do not describe completely a composition function. The main problem of such an operator generally concerns the aliasing problem. If δ_0 reads a location l_0 and δ_1 writes into the location l_1, two cases should be considered when computing $\delta_0 \oplus \delta_1$: either l_0 and l_1 are aliases or they are not. Consequently $\delta_0 \oplus \delta_1$ should contain a conditional testing the aliasing of l_0 and l_1. Hence, multiple compositions may result in huge functions (exponential blow-up). Therefore, the use of a *simplification function* is required to keep the STFs "small". Let **isalias**(x,y) denote the dynamic aliasing test of locations x and y: the condition **isalias**(x,y) is true if and only if x and y represent the same memory location. Then, the following simplification rules are acceptable:

$$\begin{array}{rl} \lfloor \textbf{isalias}(x,x)\ ?\ \delta_t \mid \delta_f \rfloor \rightarrow \delta_t & \\ \lfloor \textbf{isalias}(x,r_i)\ ?\ \delta_t \mid \delta_f \rfloor \rightarrow \delta_f & \text{if } x \text{ is a variable} \\ \lfloor \textbf{isalias}(t[e_0],t[e_1])\ ?\ \delta_t \mid \delta_f \rfloor \rightarrow \delta_t & \text{if } \forall \rho,\ [\![e_0]\!](\rho) = [\![e_1]\!](\rho) \end{array}$$

DEFINITION 2 (SIMPLIFICATION FUNCTION). *A* simplification function *is a computable function $\mathcal{S} : \mathbb{T} \rightarrow \mathbb{T}$ which preserves the semantics of STFs: $\forall \delta \in \mathbb{T},\ [\![\mathcal{S}(\delta)]\!] = [\![\delta]\!]$*

3.2 Semantics Using Symbolic Transfer Functions

In the following, we always assume that the transfer relation of a program can be encoded as a table of STFs. Let $P = (L_P, \rightarrow_P, i_P)$. A table of STFs encoding the transfer relation $\rightarrow_P$ is a family $\Delta_P = (\delta_{l,l'})_{l,l' \in L_P}$ where $\delta_{l,l'}$ stands for the STF between points l and l' and is such that:

$$\delta_{l,l'}(\rho) = \rho' \iff (l,\rho) \rightarrow_P (l',\rho')$$

Intuitively, $\delta_{l,l'}(\rho) = \rho'$ means that program point l' can be reached with store ρ' after one transition step from the state (l,ρ) if $\rho' \neq \bot_S$. If $\delta_{l,l'}(\rho) = \bot_S$, then l' cannot be reached after the execution state (l,ρ). Therefore, the relation $\rightarrow_P$ is completely and uniquely characterized by Δ_P (this also holds in the non-deterministic case). In practice, most of the $\delta_{l,l'}$ are $\Box$, so we use a sparse representation for the transition tables.

In the case of C programs, each statement defines a family of transfer functions; in case of assembly programs, each instruction defines one or two STFs (two in the case of conditional branching only). As an example, we give the STF-based encoding for two instructions:

- The *load* l : `load` $r_0, \underline{x}(v)$; l' : ... succeeds if and only if $\underline{x}+v$ is a valid address:

$$\delta_{l,l'} = \lfloor \textbf{isaddr}(\underline{x}+v)\ ?\ \lfloor r_0 \leftarrow \textbf{content}(\underline{x}+v) \rfloor \mid \Box \rfloor$$

- The *comparison* l : `cmp` r_0, r_1; l' : ...:

$$\begin{array}{rl} \delta_{l,l'} = \lfloor r_0 < r_1\ ?\ \lfloor cr \leftarrow LT \rfloor & \\ \quad \mid \lfloor r_0 = r_1\ ?\ \lfloor cr \leftarrow EQ \rfloor \mid \lfloor cr \leftarrow GT \rfloor \rfloor \rfloor & \end{array}$$

For a complete definition of the semantics of source (Fig. 1(a)) and assembly (Fig. 1(b)) programs using STFs, we refer the reader to Appendices A and B.

Henceforth, we assume that the semantics of programs is given by their STFs-based representation. The STFs-based assembly semantics may abstract away some "low level" aspects of assembly instructions, like the problems of alignments inherent in usual store and load instructions or like the instructions devoted to the management of the cache memory and of the interruptions mechanism (system-level operations). Similarly, the semantics encoding may ignore overflows. If so, the forthcoming proofs of correctness do not take these properties into account, because of this initial abstraction. The assembly model is defined by the translation of assembly programs in tables of STFs.

3.3 Towards Compilation Certification

We assume that the compiler provides the program point mapping and the memory location mapping together with the program P_a when compiling P_s. In the following, we suppose that $L_s^r = L_s$ and $X_s^r = X_s$ without a loss of generality; hence, Φ_s is the identity. We envisage here the definition of a new program observationally equivalent to P_a and more adapted to further formal reasoning about compilation. This program (the *reduced-LTS*) will be widely used in Sect. 4. A source reduced-LTS is usually also defined.

Reduction of Labeled Transition Systems: We suppose here that L_a^r contains at least one point in each strongly connected component of P_a: we assume that at least one point in each loop of the compiled program can be mapped to a point in the source, in order to be able to relate in a good way the behaviors of both. The entry point i_a is also supposed to be in L_a^r. Then, a *reduced-LTS* P_a^r can be defined by considering the points in L_a^r only and defining adequate STFs between these points. If $l, l' \in L_a^r$ and $c = l, l_0, \ldots, l_n, l'$ is a path from l to l' such that $\forall i, l_i \notin L_a^r$, then we can define the associated STF $\delta_{l,l'}^c$ by composing single step STFs (Th. 1). There is only a finite number of paths c satisfying the above condition. The transition relation of the reduced-LTS is defined by $(l,\rho) \rightarrow (l',\rho') \Longleftrightarrow \exists c,\ \delta_{l,l'}^c(\rho) = \rho'$. Note that any pair of points of the reduced-LTS defines a finite set of STFs instead of one STF in the case of the original LTS. The traces of the reduced-LTS are the same as those of the initial LTS modulo deletion of intermediate states, as mentioned below:

THEOREM 2 (LTS REDUCTION). *Let Ψ_r be the function $\Psi_r : (L_a \times S)^\star \rightarrow (L_a^r \times S)^\star$; $\langle (l_0,\rho_0), \ldots, (l_n,\rho_n) \rangle \mapsto \langle (l_{k_0},\rho_{k_0}), \ldots, (l_{k_m},\rho_{k_m}) \rangle$ where $\{k_0, \ldots, k_m\} = \{i \mid l_i \in L_a^r\}$ and $k_0 < \ldots < k_m$. Then, Ψ_r is onto between $[\![P_a]\!]$ and $[\![P_a^r]\!]$ (the* program point forget *operator): $\forall \sigma' \in [\![P_a^r]\!],\ \exists \sigma \in [\![P_a]\!],\ \sigma' = \Psi_r(\sigma)$.*

If the compilation of P_s into P_a is correct, the bijection between traces of both programs maps a one-step transition in P_s to a one-step transition in P_a^r.

Simplification: The computation of the reduced-LTS should benefit from the use of a simplification. For instance, real assembly languages compute and load addresses in several stages: the first half of the address is loaded, then the second half is loaded and shifted and the sum yields the address. Similarly, the load of a double floating point constant can be split into several operations. A simplification (Def. 2) reduces such sequences of operations into atomic ones. In practice, we noted that using an efficient simplification procedure was crucial to make further certification easier.

4 Certification of Compilation and of Compiled Programs

We consider the certification of compilation and of compiled programs in the framework introduced above.

4.1 Abstract Invariant Translation and Invariant Checking

A first goal is to check that the assembly programs complies with some safety requirements. This can be done by inferring an invariant of the source program, translating it into a property of the assembly program and checking the soundness of this invariant. We

envisage here a class of invariants (large enough for most applications) and the way we can translate them, using the formalism of Abstract Interpretation [8].

Let $D^\sharp$ be an abstract domain for representing sets of stores: $(\mathbb{P}(S),\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp,\sqsubseteq)$. The domain $D^\sharp$ provides usual abstract operations ($\bot$ and $\top$ constants, an associative and commutative $\sqcup$ operator and $\sqcap,\Delta,\nabla$ operators), a sound assignment operator (**assign** $:\mathbb{P}(X)\times E\times D^\sharp \to D^\sharp$ inputs a set of locations, an expression and an abstract invariant and handles the assignment; in case the set of locations is not a singleton, we are in presence of a "may assign"), a sound guard operator (**guard** $: C\times\mathbb{B}\times D^\sharp \to D^\sharp$ inputs a condition, a boolean and an abstract invariant). The soundness of the guard operator boils down to $\forall c\in\mathtt{C},\ \forall d\in D^\sharp,\ \forall b\in\mathbb{B},\ \{\rho\in\gamma(d)\mid [\![c]\!](\rho)=b\}\subseteq\gamma(\mathbf{guard}(c,b,d))$: intuitively $\gamma(\mathbf{guard}(c,b,d))$ contains at least all the stores in $\gamma(d)$ which evaluate the condition c to b (the soundness of **assign** is similar). It is a general result of the Abstract Interpretation theory [8] that a sound approximation $[\![P_s]\!]^\sharp : L_s \to D^\sharp$ of the behaviors of the source program can be computed: $\forall l\in L_s,\ \{\rho\in S\mid\langle\ldots,(l,\rho),\ldots\rangle\in[\![P_s]\!]\}\subseteq\gamma([\![P_s]\!]^\sharp(l))$.

The issue we address here is how to derive from $[\![P_s]\!]^\sharp$ a property for the assembly program, i.e. a local invariant $I^\sharp_a(l_a)$ for each point $l_a\in L_a$. The compilation of P_s into P_a is here assumed to be correct. We write $\delta^\sharp$ for $\alpha\circ[\![\delta]\!]\circ\gamma$ (most precise abstract transfer function corresponding to δ; an upper approximation of it is usually computed).

- Let $l_s\in L_s$, $l_a\in L^r_a$ such that $\pi_L(l_s)=l_a$ and $\rho_a\in S$ such that there exists $\sigma_a=\langle\ldots,(l_a,\rho_a),\ldots\rangle\in[\![P_a]\!]$. The correctness of the compilation entails the existence of $\rho_s\in S$ such that $\rho_s\simeq_{\pi_X}\rho_a$. We deduce that $\rho_a\circ\pi_X\in\gamma([\![P_s]\!]^\sharp)(l_s)$, which defines a sound local invariant $I^\sharp_a(l_a)$ at the assembly level at point l_a. Intuitively, a property of x_s is "inherited" by $\pi_X(x_s)$ and if $\neg(\exists x_s\in X^r_s,\ x_a=\pi_X(x_s))$, then the source invariant $[\![P_s]\!]^\sharp(l_s)$ does not give any information about x_a.
- Once a local invariant has been determined for all the points of the reduced-LTS, a local invariant can be computed for any other label of L_a under the assumption that any loop of P_a contains at least a point that is in L^r_a. Indeed, let $l_a\in L_a$. There exists a finite set $\mathcal{P}$ of paths $c=l'_a,l_0,\ldots,l_n,l_a$ such that $l'_a\in L^r_a$ and $l_0\notin L^r_a,\ldots,l_n\notin L^r_a$ and we can define a local invariant $I^\sharp_a(l_a)$:

$$I^\sharp_a(l_a)=\bigsqcup_{c\in\mathcal{P},c=l'_a,\ldots,l_a}\delta^\sharp_c(I^\sharp_a(l'_a)). \tag{1}$$

  where $\delta^\sharp_c=(\delta_{l_n,l_a}\oplus\ldots\oplus\delta_{l'_a,l_0})^\sharp$ if $c=l'_a,l_0,\ldots,l_n,l_a$

Moreover, $I^\sharp_a(l_a)$ is sound: $\gamma(\{\rho\in S\mid\langle\ldots,(l_a,\rho),\ldots\rangle\in[\![P_a]\!]\})\sqsubseteq I^\sharp_a(l_a)$. We deduce that a sound invariant can be derived for P_a from $[\![P_s]\!]^\sharp$ (see [20] for an extensive proof):

THEOREM 3 (INVARIANT TRANSLATION SOUNDNESS). *If the compilation of P_s into P_a is correct, then the invariant $I^\sharp_a$ is sound, that is:*

$$\forall l_a\in L_a,\ \{\rho\in S\mid\langle\ldots,(l_a,\rho),\ldots\rangle\}\subseteq\gamma(I^\sharp_a(l_a)).$$

Assuming the correctness of the compilation of P_s into P_a is generally not acceptable; hence, we envisage now a way to check the soundness of the translated invariant $I^\sharp_a$ independently of the way it is derived from the source invariant.

Intuitively, Invariant Checking amounts to verifying the inductiveness of the translated invariant at the reduced-LTS level (Th. 2). The principle of Invariant Checking (IC) is stated by the following theorem (we still suppose that L^r_a contains one point in each cycle of L_a):

THEOREM 4 (INVARIANT CHECKING). *If $I^\sharp_a$ is an assembly abstract invariant such that for all $l_a,l'_a\in L^r_a$, and for all path c : $l_a,l_0,\ldots,l_n,l'_a$ such that $l_0\notin L^r_a,\ldots,l_n\notin L^r_a$, $\delta^\sharp_c(I^\sharp_a(l_a))\sqsubseteq I^\sharp_a(l'_a)$, then $I^\sharp_a$ is sound.*

EXAMPLE 2. *A basic interval analysis would infer the invariants $i\in[0,n-1]$ at point l^s_3 and $i\in[1,n]$ at point l^s_4. The translated invariant $I^\sharp_a$ is such that $I^\sharp_a(l^a_8)(\underline{i})=[0,n-1]$ and $I^\sharp_a(l^a_{11})(\underline{i})=[1,n]$. This ensures local stability (for l^a_8 and l^a_{11}) since $\delta^\sharp_{l^a_8,l^a_{11}}(I^\sharp_a(l^a_8))\sqsubseteq I^\sharp_a(l^a_{11})$. Stability is proved by checking similar inequalities for all pairs of program points in the assembly reduced-LTS.*

In practice, this checking is performed in an abstract domain $D'^\sharp$ more precise than $D^\sharp$, since the structure of the assembly program can be much more complicated and require the refinement of $I^\sharp_a$. For instance, the checking of an interval invariant at point l^a_{15} in the example above requires knowing that the content of r_0 and the content of the memory cell of address $\underline{i}$ are equal at point l^a_{11}. This information is provided by an auxiliary analysis using the abstract domain D_E of the set of the partitions of the set of the assembly memory locations. This domain defines a Galois connection [8] $(\mathbb{P}(X_s\to R),\subseteq)\underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}}(D_E,\sqsubseteq_e)$, where:

$$\rho\in\gamma_e(\Upsilon)\iff\forall\psi\in\Upsilon,\ \exists v\in R,\ \forall x\in\psi,\ \rho(x)=v.$$

Then, the invariant checking would be done in the reduced product of $D^\sharp$ and D_E [10] (usually, only an approximation of the reduced product is effectively computed). Among the other domain refinements which may turn out to be necessary to handle a real assembly language, we can cite the reduced product with the congruence domain [11]: the pair $(a,b)\in\mathbb{N}\times\mathbb{N}$ represents the set $\{a+kb\mid k\in\mathbb{Z}\}$. Indeed, analyzing the access to struct members or array cells requires congruence information about the memory addresses.

Precision and STFs: The decomposition of source statements into sequences of instructions induces a loss of structure which may compromise invariant checking when STFs are not used. Relational abstract domains often handle precisely complex operations (assignments and guards of complex expressions) when done in one step as is the case for the octagons [13] for some linear assignments like $y:=\Sigma_i a_i\star x_i$ where $a_i\in\mathbb{Z}$. Similarly, precise transfer functions carry out in a better way "big" expressions in one step. For instance, the symbolic manipulations implemented in the analyzer of [5] reduce $x+(y-x)$ to y (under certain conditions and with corrections due to possible floating points rounding errors), which may not be doable anymore if an assignment is split into a sequence of statements. More formally, if $\zeta=[\![\delta_0]\!]\circ\ldots\circ[\![\delta_n]\!]$ then $\zeta^\sharp\sqsubseteq\delta^\sharp_0\circ\ldots\circ\delta^\sharp_n$ since $\lambda x.x\sqsubseteq\gamma\circ\alpha$. Generally, $\zeta^\sharp\sqsubset\delta^\sharp_0\circ\ldots\circ\delta^\sharp_n$: this strict inequality corresponds to a loss of precision. However, STFs solve this problem since they rebuild expressions (Th. 1); hence, they allow to recover the precision achieved at the source level, provided the abstraction occurs *after* symbolic composition of the STFs [7], i.e. provided we use $(\delta_0\oplus\ldots\oplus\delta_n)^\sharp$, which is more precise than $\delta^\sharp_0\circ\ldots\circ\delta^\sharp_n$.

REMARK 1 (REFINEMENT AND STFS). *The use of STFs can supersede the use of a symbolic domain or some domain refinement. For instance, the checking procedure of [20] requires a partitioning over the values of the condition register to handle branching. This*

refinement is not necessary any more when using STFs, which is a major improvement compared to [20] (lower memory usage).

4.2 Translation Validation

The goal of Translation Validation is to prove each compilation separately by showing that the assembly program correctly implements the source program. The most common approach proceeds to this equivalence proof locally.

As seen in Sect. 2.1, the semantics of the program P_i boils down to the least fixpoint of an operator F_{P_i} computed in a complete lattice (for $i \in \{s,a\}$). By continuity of F_{P_i} for the inclusion order, $[\![P_i]\!] = \bigcup_{n\in\omega} F_{P_i}^n(\emptyset)$. Since the erasure operators Φ_i and $\simeq$ are also continuous (for the inclusion order) and $\Phi_i(\emptyset) = \emptyset$, we deduce the following theorem, which states the principle of Translation Validation:

THEOREM 5 (TRANSLATION VALIDATION). *If for all $E_s \in \mathbb{P}((L_s \times S_s)^\star)$ and $E_a \in \mathbb{P}((L_a \times S_a)^\star)$, $\Phi_s(E_s) \simeq \Phi_a(E_a) \Longrightarrow \Phi_s(F_s(E_s)) \simeq \Phi_a(F_a(E_a))$, then $\Phi_s([\![P_s]\!]) \simeq \Phi_a([\![P_a]\!])$.*

The similarity of Theorem 5 with a Fixpoint Transfer Theorem [9] can be noticed (the proofs are identical).

In practice, the implication $\Phi_s(E_s) \simeq \Phi_a(E_a) \Longrightarrow \Phi_s(F_s(E_s)) \simeq \Phi_a(F_a(E_a))$ is proved by showing the following property over the assembly reduced-LTS, for all program points l_s, l'_s, l_a, l'_a such that $\pi_L(l_s) = l_a$ and $\pi_L(l'_s) = l'_a$:

$$\begin{array}{l}\forall \rho_s \in X_s \to R,\ \forall \rho_a \in X_a \to R,\\ \quad \rho_s \simeq_{\pi_X} \rho_a \ \Rightarrow [\![\delta_{l_s,l'_s}]\!](\rho_s) \simeq_{\pi_X} [\![\delta_{l_a,l'_a}]\!](\rho_a)\end{array} \quad (2)$$

This equivalence of STFs can be proved by a conservative decision procedure at the reduced-LTS level (Sect. 3.3), by observing only the memory locations in π_X. A slight generalization of (2) is used in practice, since there are generally several STFs between two control points of a Reduced-LTS, as seen in Sect. 3.3. Handling non-determinism would require considering sets of states instead of states in (2). In case the semantics of the source language leaves some behaviors undefined (as the C semantics does), the proof of adequation of STFs outlined by (2) should restrict to well-defined behaviors: the definition of STFs can be extended by an additional "undefined behavior" STF $\circledast$, such that $\forall \rho \in S,\ [\![\circledast]\!](\rho) = S$. Another approach to this problem proceeds by specializing the source semantics, which is doable if the way the compiler handles the "undefined" cases is known.

EXAMPLE 3 (STFS ADEQUATION). *In the case of the example programs of Fig. 2, l_2^s corresponds to l_4^a; l_3^s to l_8^a. After automatic simplification, $\delta_{l_2^s,l_3^s} = \lfloor i < n\ ?\ \imath \mid \square \rfloor$ and $\delta_{l_4^a,l_8^a} = \lfloor \mathbf{content}(\underline{i}) < n\ ?\ \lfloor cr \leftarrow LT, r_1 \leftarrow n, r_0 \leftarrow \mathbf{content}(\underline{i}) \rfloor \mid \square \rfloor$ (since $\mathbf{isaddr}(\underline{i})$ holds). These STFs are symbolically equivalent (in the sense of (2)), since cr, r_0 and r_1 are not mapped to any source variable and since the location of address $\underline{i}$ is mapped to the variable i.*

Checking the equivalence of $\delta_{l_3^s,l_4^s}$ and $\delta_{l_8^a,l_{11}^a}$ requires to assume that $\underline{t} + \mathbf{content}(\underline{i})$ is a valid address if and only if i is a valid index in the array t (predicate noted $\mathbf{isindex}(t,i)$) and that these locations are associated by π_X: this stems from π_X and from the assembly memory model.

As in Ex. 2, the case of node l_5^s requires more information: checking the equivalence of $\delta_{l_4^s,l_5^s}$ and $\delta_{l_{11}^a,l_{15}^a}$ requires knowing that r_0 and $\underline{i}$ store the same value at point l_{11}^a, which can be proved by an auxiliary equality analysis (as in Sect. 4.1).

Principle of a translation verifier: In practice, the local property (2) is checked by a specialized theorem prover, able to show the equivalence of expressions and to decide implications and incompatibilities between conditions. First, we observe that STFs can be seen as trees. A branch in an STF defines a (possibly void) sequence of conditions $c_0, \ldots, c_n$ (when an "else" branch is taken, we get a negative condition) and either a (possibly empty) set of assignments or an error transition $\square$. We propose here a typical algorithm for attempting to prove the local property displayed above (l_s, l'_s, l_a, l'_a denote program points such that $\pi_L(l_s) = l_a$ and $\pi_L(l'_s) = l'_a$):

- Assume that corresponding source and assembly memory locations contain the same value.
- Simplify the trees δ_{l_s,l'_s} and δ_{l_a,l'_a} by eliminating redundant conditions on their branches (and using equivalences like $\lfloor c\ ?\ \square \mid \square \rfloor \equiv \square$).
- Reduce the assembly STF using information about equality relations provided by a preliminary analysis.
- For each pair of branches $c_0^s, \ldots, c_m^s$ of leaf f^s in δ_{l_s,l'_s} and $c_0^a, \ldots, c_n^a$ of leaf f^a in δ_{l_a,l'_a}, assume $c_0^s, \ldots, c_m^s, c_0^a, \ldots, c_n^a$ and check that:
  - either the current assumption is contradictory (the branches are not compatible);
  - or the leaves f^s and f^a are equivalent: either they are both the $\square$ STF or they correspond to semantically equivalent sequences of assignments.

If one step fails to prove the incompatibility of the branches or the equivalence of the assignments at the leaves, then the Translation Validation fails and yields an (possibly false) alarm. This algorithm is conservative and possibly not optimal (global simplifications could be done right at the beginning by considering compatible branches only).

Among systems proving a semantic equivalence of source and compiled programs we can cite the systems of [19, 23] and of [17].

4.3 Translation Validation-based Invariant Translation

TV proves the equivalence of the source program and of the compiled program for a concrete semantic interpretation of STFs. This verification is done locally, by checking the semantic equivalence of source and assembly STFs (Th. 5). By contrast, IC (Th. 4) is based on an abstract semantic interpretation of STFs and involves the effective checking of a fixpoint. In case no alarm appears, the proof is validated. If IC or TV succeeds, the confidence level in the compilation correctness increased: both approaches aim at showing that some property of the original program is preserved.

Furthermore, TV and IT can be advantageously combined: indeed, Th. 3 suggests that invariant checking could be discarded if we can assume compilation correctness, which can be proved by TV. If two symbolic transfer functions δ and δ' are semantically equivalent, then they can be abstracted to a same abstract function $\delta^\sharp$. We deduce the principle of this approach from Th. 3 and Th. 5 as follows (with the notations of Sect. 4.1 and Sect. 4.2):

THEOREM 6. *If the hypotheses of Th. 5 hold (i.e. if for all $E_s \in \mathbb{P}((L_s \times S_s)^\star)$ and $E_a \in \mathbb{P}((L_a \times S_a)^\star)$, $\Phi_s(E_s) \simeq \Phi_a(E_a) \Longrightarrow \Phi_s(F_s(E_s)) \simeq \Phi_a(F_a(E_a))$) and if $\forall l \in L_s,\ \{\rho \mid \langle \ldots, l, \rho), \ldots \rangle\} \in [\![P_s]\!] \subseteq \gamma([\![P_s]\!]^\sharp(l))$, then the invariant $I_a^\sharp$ computed as in Sect. 4.1 is sound.*

A first advantage of doing TV instead of IC to justify the soundness of the invariant translation is that the domain refinement evoked in Ex. 2 is not necessary any more: the design of the reduced-product $D^\sharp \times D_E$ can be avoided. Moreover, running TV once is enough to prove several IT passes: only one checking procedure is required.

However, the main practical advantage we noticed is that the TV process described in Sect. 4.2 provides very useful information about the meaning of some assembly sequences. For instance, the conversion of integers into floating point values involves complicated sequences of assembly operations: in the case of the real PPC language, such a conversion is commonly compiled into a sequence of bitwise operations, subtractions and multiplication. This sequence can be proved (by hand) to be equivalent to the conversion. Then, TV should recognize such sequences and replace them by the atomic "conversion" operation: TV produces a new, simpler assembly STF equivalent to the original one. This latter STF makes the further invariant propagation step (following (1)) more efficient and more precise: indeed, the design of a precise abstract counterpart for the original intricate sequence would be painful.

Finally, we propose the following process for certifying both the compilation and the compiled code:

1. compilation of P_s and P_a to reduced-LTSs, using STFs
2. Translation Validation (Sect. 4.2); replacement of assembly STFs by "simpler" ones during the proof;
3. translation of the invariant computed at the source level as seen in Sect. 4.1 (no Invariant Checking required);
4. checking that the safety requirements are fulfilled using the translated invariant

Note that the step 4 is still strictly necessary to prove the soundness of the compiled code (i.e. that it yields no run-time error). Indeed, the equivalence proof of step 3 holds with respect to the mappings π_X and π_L: it entails that part of the computations done at the assembly level implement the source program but it does not mean that *any* computation done at the assembly level is the counterpart of some part of the source program. Therefore, the verification of the soundness of the assembly program must eventually be done at the assembly level, even if the source analysis concludes that the source program is semantically safe.

4.4 Implementation and Results

We implemented a certifier following the approach introduced in Sect. 4.3 in OCaml [12]: the IT is preceded by a TV step which allows to deal with simplified assembly STFs when translating invariants and to avoid coping with abstract invariant checking.

Our goal was to certify automatically both the compilation and the absence of Run-Time Errors (RTE) in the compiled assembly programs. We also expected the certifier to scale up. The target architecture is a 32 bits version of the Power-PC processor; the compiler is **gcc** 3.0.2 for Embedded ABI (cross-compiler). The source invariants are computed using the analyzer presented in [5] and achieve a very low false alarms number when used for checking RTE. This result is achieved thanks to a very precise domain, using octagons [13], boolean relations, ellipsoid domain, control-based partitioning and other domain refinements.

The benchmarks involve three programs (of small, medium and large size) corresponding to typical embedded applications: the third one is a real life application, running in true systems for a long time (it amounts to about 75 kLOC after constant propagation and features 10000 static variables) whereas the two others correspond to representative development examples. These programs are usually compiled with a low level of optimization since the user generally wishes the compiler to produce regular and predictable results in the context of critical systems. Hence, the certification we attempt to operate here is realistic.

The translation verifier handles most C features (excluding dynamic memory allocation through pointers which is not used in the family of highly critical programs under consideration): procedures and functions, structs, enums, arrays and basic data types and all the operations on these data-types. The fragments of C and of the Power PC assembly language we considered are larger than those presented in Appendices A and B: In particular, a restricted form of alias is handled for the sake of passing by reference of some kind of function arguments like arrays. Non-determinism is also accommodated (volatile variables). The mappings π_X (for variables) and π_L (for program points) are extracted from standard debugging information. The verifier uses the Stabs format (hence, it inputs assembly programs including these data).

Program	1	2	3
Size of the source (lines)	369	9500	74365
Size of the assembly (lines)	1932	56626	344005
(1) C frontend (s)	0,04	0,53	2,97
(1) Assembly frontend (s)	0,08	0,97	13
(1) Mapping construction (s)	0,03	0,39	0,81
(2) Translation Validation (s)	0,14	0,62	9,45
Alarms in Trans. Validation	0	0	0
(3) Invariant translation (4) + RTE checking (s)	0,23	8,22	84,5
Alarms (Runtime Errors)	0	0	22
Source analysis (s)	1,15	46,79	3698

Table 1. Benchmarks

The equivalence proof is carried out by an optimized first order Resolution-based prover which accepts only clauses of length 1 (details about Resolution can be found in [6]). This requires a pre-normalization of STFs (the conditions are all expanded into atomic ones). Clauses are all of the form $e_0 \mathcal{R} e_1$ (where $\mathcal{R}$ is a relation) or e_0 (where e_0 is a boolean expression); hence, in case a derivation of false exists, it can be found immediately. The Unification procedure is mostly based on the ML pattern matching of the rules specifying the equivalence of expressions; it is not complete but very fast. Two auxiliary analyses were necessary in order to make the translation validation successful:

- the equality analysis mentioned in Sect. 4.1;
- a congruence analysis [11] since alignments should be handled carefully (in order to accommodate struct and enum data types).

The whole development amounts to about 33000 lines of OCaml code: The various parsers and interfaces (e.g. with the source analyzer) are about 17000 lines; the kernel of the certifier (the implementation of the STFs and the prover) is about 6000 lines; the symbolic encoding functions (i.e. the formal definition of the semantics of the source and assembly languages) are about 3000 lines; the invariant translator and the certifier are about 5000 lines. The most critical and complicated part of the system corresponds to the symbolic composition (2000 lines) and to the prover (1500 lines).

The four steps described in the end of Sect. 4.3 were run on a 2.4 GHz Intel Xeon with 4 Gbytes of RAM. TV succeeds on the three programs (no alarm is raised). The results of the benchmarks are given in table 1 (sizes are in lines, times in seconds).

As can be seen in the table above, most of the time for the assembly certification is spent in the invariant translation and the checking of RTE which is due to the size of the invariants dumped by the source analyzer (although only part of the source invariant was used for the assembly certification). The step (1) is longer than TV because the frontend performs massive STFs simplifications, which make the equivalence checking more easy: there was no dynamic aliasing tests (Sect. 3.1) in the simplified STFs. The checking time is rather

small compared to the source analysis done to synthesize, process and serialize the source invariant (last raw). The memory requirement for the third program is 750 Mbytes for the source analysis; 400 Mbytes for the assembly certification (i.e. for the step (4)). This requirements are much smaller than those of the IC algorithm used in [20], since the certification of the first program was about 20 s long and required about 80 Mbytes; the third program could not be treated. The key of scalability is to make IC redundant and to do TV instead: the abstract fixpoint checking turns out to be much too resource consuming. We believe that a direct analysis of the assembly program would be very painful to design (complex invariants) and not scalable.

TV is successful: no false alarm is raised by the equivalence proofs, which justifies the further IT. No possible runtime errors are reported in the first two programs. The alarms reported for the last one concern the same program points as the source analysis for the RTE checking. These alarms can be disproved by hand, so the certifier proves able to certify the assembly program almost completely. The main conclusion of this experimental study is that the approach to IT based on TV (Sect. 4.3, Th. 6) is realistic and scalable to large critical applications.

5 Optimizing Compilation

The previous sections focused on non-optimizing compilation, as defined in Def. 1. We envisage the extension of our framework to optimizing transformations by considering a few representative cases and generalizing the solutions.

5.1 Methodology

First, we notice that most optimizations do not fit in the above framework since they are not considered correct in Def. 1. Hence, this definition should be generalized, so as to accept these new transformations as "correct compilations".

The certification methods presented in Sect. 4 are based on the reduced-LTS notion. Hence, two approaches for handling an optimizing transformation can be proposed:

- enhancing the definition of reduced-LTSs so as to extend the certification method (TV and IC algorithms change);
- integrating the proof of correctness of the optimization into the computation of the reduced-LTS, which allows to extend the certification method straightforwardly: the optimization is proved first and a "de-optimized" reduced-LTS is derived; then compilation is proved in the previous sense (only IT must be adapted since π_X and π_L are modified).

In the following, we address some families of optimizations; the issue of certifying series of optimizations is considered afterwards. Extensive references about optimizations can be found in [1, 3]. We do not claim to be exhaustive (which would not be doable here); yet, the methods presented below are general and can be extended to more involved transformations than the mere compilation considered in Sect. 4. We examplify these methods on representative members of several classes of optimizations.

5.2 Code Simplification: Dead Code/Variable Elimination

Dead-Code Elimination: In case a compiler detects that some piece of the source code is unreachable (typically after constant propagation), it may not compile it: then, some source program points are not mapped to any assembly program point. This optimization is correct in the sense of Def. 1, since correctness was defined by the data of a bijection between *subsets* of the source and assembly program points $L_s^r \subseteq L_s$ and $L_a^r \subseteq L_a$: eliminated program points should not appear in L_s^r. Non-eliminated points are expected to appear somewhere in the mapping, therefore we may envisage to check that the source program points that do not belong to L_s^r actually are dead-code (optimization checking). Dead-code analyses done by compilers are most often quite simple, so this should be checked by a rather coarse dead-code analysis. Hence, no extension of the reduced-LTS notion is required (the transformation is verified at the source reduced-LTS elaboration level).

Dead-Variable Elimination: Most compilers do liveness analysis in order to avoid storing dead-variables. A slightly more complicated definition of the variable mapping π_X can accommodate this further transformation. Indeed, the abstract counterpart of a source variable depends on the program point and may not exist at some points, so $\pi_X : L_s^r \times X_s \to X_a \cup \{\emptyset\}$ where $\pi_X(l_s, x_s) = \emptyset$ means that x_s has no assembly counterpart at point l_s and should not be live at that point if the optimization is correct.

Copy propagation and register coalescing: Compilers attempt to keep a variable that is used several times in a piece of code in a register and do not store it back to memory before using it again. For instance, the load instructions l_4^a, l_8^a and l_{11}^a in the example program of Fig. 2(b) could be eliminated, since the variable *i* is stored in register r_0 everywhere in the loop. The same approach as for dead-variable elimination works here.

5.3 Order Modifying Transformations: Instruction Level Parallelism

We envisage now the case of transformations which compromise the correspondence of program points; our study focuses on instruction scheduling. Instruction Level Parallelism (ILP or scheduling) aims at using the ability of executing several instructions at the same time featured by modern architectures, so as to cut down the cost of several cycles long instructions. Hardware scheduling is not a difficulty, since its soundness stems from the correctness of the processor (which should be addressed separately); hence, we consider software scheduling only ([2], Chap. 20). Software scheduling does not fit in the correctness definition presented in Sect. 2.2, since the pieces of code corresponding to distinct source statements might be inter-wound (assembly instructions can be permuted). Indeed, Fig. 3 displays an example of software scheduling: the body of the loop of the example program of Fig. 2 is optimized so as to reduce pipeline stalls. If load, store and arithmetic instructions all have a latency of one cycle, then the execution of the code of Fig. 3(b) yields one cycle stall against four in the case of the (non optimized) code of Fig. 3(a). The pieces of code corresponding to the statements $l_3^s \to l_4^s$ and $l_4^s \to l_5^s$ are inter-wound, so l_4^s has no counterpart in the optimized program. The mapping π_L is no longer defined; π_X becomes a function from $(L_s \times X)$ to $\mathbb{P}(L_a \times X)$. Before we extend the algorithms proposed in Sect. 4, we need to set up a new notion of assembly program points, so as to set up again a mapping between P_s and P_a.

Correctness of Compilation: Let us consider a source execution state of the form (l_4^s, ρ_s) in the program of Fig. 2. This state corresponds to a state of the form (l_{11}^a, ρ_a) in the program of Fig. 3(a). However, it does not correspond to a unique state in the optimized program (Fig. 3(b)): the value of $\underline{i}$ is effectively computed at point l_{14}^o, whereas the value of $\underline{t} + \underline{i}$ is read by the next statement at point l_{11}^o. Therefore, a source execution state (l^s, ρ) corresponds to a sequence of states in the optimized program; hence, we propose to define a new assembly observational abstraction α_a^r by extending the Φ_a function (Sect. 2.2). The new Φ_a should rebuild the orig-

l_8^a	load $r_0, \underline{i}(0)$	l_8^o	load $r_0, \underline{i}(0)$
l_9^a	add $r_0, r_0, 1$	l_9^o	load $r_1, \underline{x}(0)$
l_{10}^a	store $r_0, \underline{i}(0)$	l_{10}^o	add $r_0, r_0, 1$
l_{11}^a	load $r_1, \underline{x}(0)$	l_{11}^o	load $r_2, \underline{t}(r_0)$
l_{12}^a	load $r_2, \underline{t}(r_0)$	l_{12}^o	add r_1, r_1, r_2
l_{13}^a	add r_1, r_1, r_2	l_{13}^o	store $r_0, \underline{i}(0)$
l_{14}^a	store $r_1, \underline{x}(0)$	l_{14}^o	store $r_1, \underline{x}(0)$
l_{15}^a	...	l_{15}^o	...

(a) Unoptimized code (b) Optimized code

Figure 3. Software Scheduling

inal control points, by associating so-called *fictitious points* to sequences of labels in the optimized code:

DEFINITION 3 (FICTITIOUS STATE). *In case $l^s \in L_s$ corresponds to the sequence of assembly points $l_0^a, \ldots, l_n^a$, we introduce a* fictitious label l^f *representing the sequence and a set of* fictitious memory locations $X_{l_f} \subseteq L_a \times X$ *representing the memory locations observed at that point. Furthermore, we assert that $\pi_L(l^s) = l^f$.*

The fictitious state (l^f, ρ^f) *is associated to the sequence of states $(l_0^a, \rho_0^a), \ldots, (l_n^a, \rho_n^a)$ if and only if ρ^f is defined by $\forall (l_i^a, x_a) \in X_f$, $\rho^f(x_a) = \rho_i^a(x_a)$.*

Note that the definition of the fictitious states is not ambiguous: in case $\pi_X(l^s, x_s) \cap X_{l_f}$ contains more than one point, the definition of $\rho^f(x_s)$ does not depend on the choice of the corresponding fictitious location, since $\forall (l_i^a, x_a), (l_{i'}^a, x'_a) \in \pi_X(l_s, x_s)$, $\rho_i^a(x_a) = \rho_{i'}^a(x'_a)$ if π_X is correct (in practice, the equality analysis of Sect. 4.1 subsumes the verification of the correctness of π_X).

The new Φ_a operator inputs a trace σ and builds up a *fictitious trace* σ^f by concatenating fictitious states corresponding to the sequences of states in σ in the same order as they appear in σ. As in Sect. 2.2, the new α_a^r operator can be defined by $\alpha_a^r(\mathcal{E}) = \{\Phi_a(\sigma) \mid \sigma \in \mathcal{E}\}$ and the transformation is said to be correct if $\alpha_s^r([\![P_s]\!]) \simeq \alpha_a^r([\![P_a]\!])$.

EXAMPLE 4. *The mapping of the first and of the last points of the loop are unchanged: source variables i, x, t at point l_3^s correspond to assembly locations of addresses $\underline{i}, \underline{x}, \underline{t}$ at point l_8^o and the same for points l_5^s and l_{15}^o. Yet, at point l_4^s, $(l_{14}^o, \underline{i}) \in \pi_X(l_4^s, i)$; $(l_9^o, \underline{x}) \in \pi_X(l_4^s, x)$ and $(l_9^o, \underline{t}+k), \ldots, (l_{14}^o, \underline{t}+k) \in \pi_X(l_4^s, t[k])$. This mapping defines a corresponding "fictitious point" l_2^f for l_4^s. Fictitious locations are $(l_{14}^o, \underline{i})$, $(l_9^o, \underline{x})$, $(l_{11}^o, \underline{t}[k])$ and (l_{11}^o, r_i) (the last choice is arbitrary). The fictitious point l_1^f (resp. l_3^f) corresponds to the point l_8^o (resp. l_{15}^o). Fictitious locations for l_1^f are those of l_8^o and the same for l_3^f. The compilation of P_s (Fig. 2(b)) into P_o (Fig. 3(b)) is correct according to the new compilation correctness definition.*

Computing Reduced-LTS: The computation of the reduced-LTS is based on the definition of STFs between fictitious points, which is technical but easy. The TV (Th. 5) and the IC (Th. 4) algorithms are extended straightforwardly to the reduced-LTS based on fictitious points (which is a reduced-LTS in the sense of Sect. 3.3).

EXAMPLE 5. *The transfer function $\delta_{l_2^f, l_3^f}$ displayed below can be proved equivalent to $\delta_{l_3^s, l_4^s} = \lfloor \textbf{isindex}(t, i)\ ?\ \lfloor x \leftarrow x + t[i] \rfloor \mid \square \rfloor$ since the fictitious locations (l_{11}^o, r_0) and $(l_{14}^o, \underline{i})$ contain the same value as would be shown by a simple extension of the equality analysis:*

$$\delta_{l_2^f, l_3^f} = \lfloor \textbf{isaddr}((l_{11}^o, \underline{t}) + (l_{11}^o, r_0))\ ? \lfloor (l_{15}^o, r_1) \leftarrow \textbf{content}((l_{11}^o, \underline{t}) + (l_{11}^o, r_0)), (l_{15}^o, r_2) \leftarrow (l_9^o, \underline{x}) + \textbf{content}((l_{11}^o, \underline{t}) + (l_{11}^o, r_0)), (l_{15}^o, \underline{x}) \leftarrow (l_9^o, \underline{x}) + \textbf{content}((l_{11}^o, \underline{t}) + (l_{11}^o, r_0)) \rfloor \mid \square \rfloor$$

IC is similar (the same STFs are interpreted in the abstract domain instead of symbolically).

Invariant Translation: The IT algorithm (Th. 3) should be adapted so as to produce results for the assembly LTS (and not only for the reduced-LTS): a fictitious point is a label of the reduced-LTS but *not* of the assembly LTS. Indeed, the above extension of Def. 1 allows to derive a local invariant for each point of the reduced-LTS only (i.e. for each fictitious point).

The solution to this problem also comes from the STFs formalism. Let $l_a \in L_a$; let us compute a local invariant for l_a. If l^f is a fictitious point corresponding to the sequence $l_0^a, \ldots, l_n^a$ such that $\pi_X(l_s) = l^f$, a sound local invariant $I^\sharp(l^f)$ can be derived for l^f. Then, if there exists a path $c = l_0^a, \ldots, l_n^a, l_{n+1}^a, \ldots, l_m^a, l_a$, which does not encounter any other fictitious states of the reduced-LTS a sound abstract upper-approximation of the set of stores $\{\rho \mid \langle \ldots, (l_0^a, \rho_0^a), \ldots, (l_n^a, \rho_n^a), (l_{n+1}^a, \rho_{n+1}^a), \ldots, (l_m^a, \rho_m^a), (l_a, \rho) \rangle \in [\![P_a]\!]\}$ can be obtained by applying the abstract STF between l^f and l_a along the path c to $I^\sharp(l^f)$ (this STF can be computed in the same way as an STF between fictitious states). A sound invariant $I^\sharp(l_a)$ can be derived by merging these abstract upper-approximations (as in Sect. 4.1).

The same ideas apply to other optimizations which move locally some computations, like code motion, Partial Redundancy Elimination (with a more general equality analysis than in Sect. 4.1) or Common Subexpressions Elimination.

5.4 Path Modifying Transformations: Loop Unrolling

Most compilers carry out structure modifying optimizations such as loop unrolling and branch optimizations (these transformations reduce time in branchings and interact well with the scheduling optimizations considered in Sect. 5.3). These transformations break the program point mapping π_L in a different way: one source point may correspond to several assembly points (not to a sequence of points). In this section, we focus on loop unrolling. This optimization consists in grouping two successive iterations of a loop, as is the case in the example below (we use source syntax for the sake of convenience and concision; $i{+}{+}$ represents $i := i+1$):

l_0 :	$i := 0;$	l'_0 :	$i := 0;$
l_1 :	**while**$(i < 2n)\{$	$l'_{1,e}$:	**while**$(i < 2n)\{$
	l_2 : $B;$		$l'_{2,e}$: $B;$ $l'_{3,e}$: $i{+}{+};$
	l_3 : $i{+}{+};\}$	$l'_{1,o}$	$l'_{2,o}$: $B;$ $l'_{3,o}$: $i{+}{+};\}$
l_4 :		l'_4 :	

We write P for the original code; P' for the optimized one.

Correctness of Compilation: The source point l_2 corresponds to two points in P' ($l'_{2,e}$ and $l'_{2,o}$); and the same for l_3. We also duplicated l_1 into $l'_{1,e}$ and $l'_{1,o}$ for the sake of the example. The index e (resp. o) is used for points where the value of i is even (resp. odd). As in Sect. 5.3, extending the compilation correctness definition reduces to defining a new Φ_a function. Indeed, Φ_a should "collapse" the pairs of duplicated points into a same point,

mapped to the original point. We write $\phi_a : L \to L$ for the program point collapsing function. For instance, in the above example $\phi_a(l'_{2,e}) = \phi_a(l'_{2,o}) = l_2$. Then, the new assembly erasure operator is defined by:

$$\Phi_a(\langle (l,\rho),\ldots,(l',\rho')\rangle) = \langle (\phi_a(l),\rho),\ldots,(\phi_a(l'),\rho')\rangle$$

The only transformation operated on P is the unrolling; consequently, the erasure operator Φ_s is the identity.

For instance, if we consider the run of P':

$$\sigma = \langle (l'_0,\rho_0),(l'_{1,e},\rho_1),(l'_{2,e},\rho_2),(l'_{3,e},\rho_3),(l'_{1,o},\rho_4),(l'_{2,o},\rho_5)\rangle$$

then:

$$\Phi_a(\sigma) = \langle (l_0,\rho_0),(l_1,\rho_1),(l_2,\rho_2),(l_3,\rho_3),(l_1,\rho_4),(l_2,\rho_5)\rangle.$$

Given $\Phi_a(\sigma) \in [\![P]\!]$, the correspondence of $\Phi_a(\sigma)$ with a trace in $\Phi_s([\![P]\!]) = [\![P]\!]$ is trivial.

Translation Validation: The above generalization of Def. 1 cannot be taken into account at the reduced-LTS level, so the IC and TV certification algorithms (Th. 4 and Th. 5) must be adapted. More precisely, a translation verifier should prove the following:

$$\begin{array}{l} \forall l_s, l'_s \in L_s,\ \forall l_a \in \pi_L(l_s),\ \forall \rho_s \in S, \\ \quad \forall \rho_a \in \{\rho \mid \langle \ldots,(l_a,\rho),\ldots\rangle \in [\![P_a]\!]\}, \\ \qquad \rho_s \simeq_{\pi_X} \rho_a \implies \exists l'_a \in \pi_L(l'_s) \text{ such that} \\ \qquad\quad [\![\delta_{l_s,l'_s}]\!](\rho_s) \simeq_{\pi_X} [\![\delta_{l_a,l'_a}]\!](\rho_a) \end{array} \tag{3}$$

Intuitively, the formula states that for any transition on an edge l_s, l'_s in the source LTS from a state (l_s,ρ_s) and for any state (l_a,ρ_a) in the target LTS in correspondence with (l_s,ρ_s) there exists a transition in the target LTS simulating the source transition. Note the additional assumption $\rho_a \in \{\rho_a \mid \langle \ldots,(l_a,\rho_a),\ldots\rangle \in [\![P_a]\!]\}$ in (3), which was not present in (2): it is due to the partitioning of source states for a same source point by several different assembly points (like $l'_{2,e}$ and $l'_{2,o}$). For instance, in the case of the example, the transfer function $\delta_{l_1,l_2} = \lfloor i < 2n\ ?\ \imath \mid \square \rfloor$ corresponds to two transfer functions: $\delta_{l_{1,e},l_{2,e}} = \lfloor i < 2n\ ?\ \imath \mid \square \rfloor$ and $\delta_{l_{1,o},l_{2,o}} = \imath$. At point $l_{1,o}$ the value of i belongs to $[1,2n-1]$; the equivalence of $\delta_{l_{1,o},l_{2,o}}(\rho)$ and of $\delta_{l_1,l_2}(\rho)$ can be proved for the stores which achieve that property. A simple auxiliary analysis may be required to collect this kind of properties.

The case of invariant checking (Th. 4) is similar.

Invariant Translation: The derivation of a sound invariant for P' from a sound abstract invariant $I^\sharp$ for P is rather straightforward, since the new definition of compilation correctness entails that $\forall l_s \in L_s,\ \forall l_a \in \pi_X(l_s),\ \{\rho \in S \mid \langle \ldots,(l_a,\rho)\rangle \in [\![P_a]\!]\} \subseteq \{\rho \in S \mid \langle \ldots,(l_s,\rho)\rangle\} \in [\![P_s]\!]$. Consequently, if $l_a \in \pi_X(l_s)$, then $I^\sharp(l_s)$ is a sound invariant for l_a.

A more precise invariant can be computed for the target program if the source analysis uses a control partitioning "compatible" with the transformation – for instance a loop unrolling in the current example.

Many optimizations which alter the branching structure of programs can be handled by doing similar generalizations. Function inlining also falls in that case.

5.5 Structure Modifying Transformations: Loop Reversal

Some optimizations focus on small and complex pieces of code and operate important reordering transformations, like loop reversal, loop interchange, loop collapsing. For instance, loop reversal modifies a loop in the following way:

$$\begin{array}{lll} l_0: & i := 0; & l'_0: \quad i := n-1; \\ l_1: & \textbf{while}(i < n)\{ & l'_1: \quad \textbf{while}(i \geq 0)\{ \\ & \quad l_2: \ B; & \quad\quad l'_2: \ B; \\ & \quad l_3: \ i := i+1;\} & \quad\quad l'_3: \ i := i-1;\} \\ l_3: & \ldots & l'_3: \quad \ldots \end{array}$$

We write P for the original code; P' for the optimized one.

Loop reversal may allow to use a specific "branch if counter is null" instruction featured by some architectures. Such a transformation is not always legal, so compilers generally check that the transformation does not break any loop carried data-dependence, which we assume here (in case the transformation would break a data-dependence, it is illegal; hence, it is not performed).

Correctness of Compilation: Loop reversal changes the order of execution of some statements, so the adaptation of the compilation correctness definition requires new Φ_a and Φ_s functions to be defined so as to "forget" the order of execution of some pieces of code. More precisely, the assignments done inside the loop are the same since the transformation does not break any data-dependence, but they may be executed in a different order. Therefore, the new observational operators should forget the order the values are computed in at some program points. Hence, Φ_s should replace a sequence σ of states inside the loop by a function ϕ from $\{l_2,l_3\} \times X$ to the set $\mathbb{P}(R)$ such that $\phi(l_2,x) = \{\rho(x) \mid \sigma = \langle \ldots,(l_2,\rho),\ldots\rangle\}$ and the same for $\phi(l_3,x)$ (the definition of Φ_a is similar). Furthermore, the value of the induction variable i is modified before and after the loop, so Φ_a and Φ_s should abstract it away. The observational semantics of P and P' is a set of sequences made of states (outside of the reversed loop) and functions defined as above (representing a run inside the reversed loop). The compilation is correct if and only if these semantics are in bijection, as in Def. 1.

Translation Validation: Loop reversal radically changes the structure of the program, so it is not possible to operate TV without resorting to a global rule, proving the correctness of the transformation, as done in [23, 24]. The reversal is legal if and only if the permutation of two iterates is always legal and the data dependences are preserved. Hence, TV should check the absence of loop-carried dependences and the following equivalence ($B[i \leftarrow x]$ denotes the substitution of the variable i by the expression x in B):

$$\forall x,y \in \mathbb{Z},\ B[i \leftarrow x];B[i \leftarrow y] \equiv B[i \leftarrow y];B[i \leftarrow x]$$

For instance, in case the code of B is encoded into an STF δ, the above equivalence can be established by the algorithm of Sect. 4.2 applied to the STFs $\delta_B \oplus \lfloor i \leftarrow y \rfloor \oplus \delta_B \oplus \lfloor i \leftarrow x \rfloor$ and $\delta_B \oplus \lfloor i \leftarrow x \rfloor \oplus \delta_B \oplus \lfloor i \leftarrow y \rfloor$.

Invariant Translation: We first remark that the IT algorithm of Th. 3 still works for the program points outside the reversed loop.

Let us consider a program point inside the loop —for instance, l_2. The correctness of compilation entails the equality $\{\rho(x) \mid \langle \ldots,(l_2,\rho),\ldots\rangle \in [\![P]\!]\} = \{\rho(x) \mid \langle \ldots,(l'_2,\rho),\ldots\rangle \in [\![P']\!]\}$, so we can determine which classes of invariants can be translated:

- Case of non-relational invariants: The invariant $[\![P]\!]^\sharp$ is non relational if it is of the form $L \to (X \to D^\sharp)$ where $D^\sharp$ is a domain for representing sets of values ($(\mathbb{P}(R),\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp,\sqsubseteq)$). Then, the soundness of $[\![P]\!]^\sharp$ boils down to $\forall l \in L,\ \forall x \in X,\ \{\rho(x) \mid \langle \ldots,(l,\rho),\ldots\rangle\} \subseteq \gamma([\![P]\!]^\sharp(l)(x))$. Hence, the above equality implies that the non-relational invariant can be translated if the compilation is correct.

- Case of relational invariants: Examples can show that relational properties may not be preserved by the translation in presence of globally reordering transformations.

Similar arguments would extend to other reordering transformations like loop interchange, coalescing.

5.6 Results

Accommodating optimizations: Our framework turns out to allow the certification of optimizing compilation, since a wide range of optimizing transformations were considered with success. Handling a new optimization generally starts with the extension of the compilation correctness definition, by tuning the observational semantics operators Φ_s and Φ_a and by generalizing the mappings π_X and π_L. Defining more abstract observational operators allows to accept more optimizations; yet, these abstractions may render the translation of some classes of invariants impossible as was the case in Sect. 5.5. The certification algorithms can be extended to a wide range of optimizations:

- Code Simplification optimizations (Sect. 5.2) are handled straightforwardly
- Path modifying transformations (Sect. 5.4) can be accommodated by generalizing the reduced-LTS notion and the certifying procedures.
- The locally order modifying transformations like scheduling (Sect. 5.3) can be certified without changing the reduced-LTS definition: the computation of reduced-LTS is more involved but the further steps of certification (TV or invariant checking) can be mostly preserved. Only IT must be adapted (in a rather systematic way).
- The case of globally reordering optimizations (Sect. 5.5) is not so satisfactory, since it involves global rules; yet such transformations should be rather localized (e.g. applied to some small loops only), so their certification should still be amenable. Moreover, this last class of optimizations is not so important as the previous ones in the case of practical C compilers (GNU **gcc**, Diab C compiler...).

Real compilers perform series of optimizations. A definition of compilation correctness for a sequence of optimizations can be derived by composing the observational abstractions corresponding to each transformation. In case the compiler provides the intermediate programs, the certification (by TV or IT) can be carried out in several steps so as to cut down the difficulty of mapping the source and target code.

Implementation and practical aspects: We had not planed to extend the certifier of Sect. 4.4 to the certification of optimized code when implementing it. However, the implementation of a new module in the frontend allowed to tackle part of the optimizations envisaged above and other similar transformations: the code simplification transformations (Sect. 5.2), the local reordering optimizations (Sect. 5.3) and some limited forms of path modifying transformations (Sect. 5.4) —i.e. branching local transformations. The new version of the certifier is effective for the first two classes of optimizations. Loop unrolling cannot be proved due to a lack of adequate information in the compiler output; localized branching transformations are certified successfully. The **gcc** compiler does not perform any reordering transformation in the sense of Sect. 5.5, so we did not attempt to certify them. Finally, the new certifier can handle code produced by **gcc** with the option -O2 (quite high optimizing level; no possibly misleading global optimization is performed; the highest level -O3 mainly adds function inlining and loop unrolling), which is rather positive given the additional implementation effort is very reasonable: the additional frontend files amount to about 3000 lines of OCaml code (the whole compilation and assembly certifier amounts to about 33000 lines). We also noted an important slowdown of the assembly frontend, mainly due to more complex interfaces and to the additional computations done to build the reduced-LTS (partial composition of STFs as mentioned in Sect. 5.3).

The main difficulty in extending the certifier consists in finding adequate debug information, so as to know which transformation has been performed. The new certifier still uses the same standard debug information format, which does not provide much data about optimizations. Moreover, these information turn out to be sometimes wrong in the case of optimizing compilation: some scopes of variables or program point mappings were wrong, so we had to fix some debug entries by hand. The precise mapping of program points also had to be partially reconstructed by applying some non trivial processing to the Stabs debugging information. The standard debugging information formats provided by most compilers are mostly aimed at debugging only and not at mapping source and compiled programs for the sake of verification. We may suggest the introduction of a more adequate debugging information format for this purpose.

Our new certifier does not use any intermediate RTL (Register Transfer Language) dump provided by the **gcc** compiler and carries out the validation in one pass. However, it seems that handling more complicated optimizations would require a several steps process to be implemented as is the case of the systems of [17] and [23, 24], which both address more elaborate optimizations.

6 Conclusion

We provided a framework for reasoning about the certification of compilation and of compiled programs. It is mainly based on a symbolic representation of the semantics of source and compiled programs and on structures to check properties of the fixpoints of the semantic operators of source and compiled programs, hence of the semantics of source and compiled programs. Existing certification methods were integrated to the framework; furthermore, a new method for certifying assembly programs was proposed, which is based on the ability to prove the semantic equivalence of programs first: this equivalence checking not only justifies IT, but also provides more precise information about the structure of assembly programs, which renders IT more efficient and accurate. In the implementation point of view, this approach proved very successful, since strong properties about very large real examples of safety critical programs were proved at the assembly level. This success was made possible by a fine description of the equivalence between source and compiled programs.

Moreover, many classes of optimizations can be successfully accommodated in our framework. Part of them are handled at the reduced-LTS elaboration level (as code simplification or locally order modifying transformations). Path modifying or globally order modifying transformations require a stronger treatment. The invariant translation result derives from a precise definition of compilation correctness, which allows to derive systematically which class of properties of the source extends to the target code.

Last, our approach allows to factor some problems out of the design of an adequate certification process. For instance, the symbolic representation of programs is shared by IT, IC and TV. In the practical point of view, some steps are also common, as is the gathering of mapping data. Note that the approach is compiler independent (no instrumentation of the compiler is required; the certifier only needs some standard information) and even architecture independent. Indeed, accommodating a new target language amounts to

implementing a new frontend compiling programs into a symbolic representation (i.e. reduced-LTS), which formalizes the semantic model of the new assembly language.

Acknowledgment: We deeply acknowledge B. Blanchet, J. Feret and C. Hymans for their comments on an earlier version. We thank the members of the Astree project for stimulating discussions: B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné and D. Monniaux.

7 References

[1] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997.

[2] A. W. Appel. Foundational Proof-Carrying Code. In *16th LICS*, pages 247–256, 2001.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[4] 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.

[5] 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 *PLDI'03*, pages 196–207, 2003.

[6] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Computer Science Classics, 1973.

[7] C. Colby and P. Lee. Trace-Based Program Analysis. In *23rd POPL*, pages 195–207, St. Petersburg Beach, (Florida USA), 1996.

[8] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, 1977.

[9] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[10] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[11] P. Granger. Static Analysis of Arithmetical Congruences. *Int. J. Computer. Math.*, 1989.

[12] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual (release 3.06). Technical report, INRIA, Rocquencourt, France, 2002.

[13] A. Miné. The Octagon Abstract Domain. In *Analysis, Slicing and Transformation 2001 (in WCRE 2001)*, IEEE, 2001.

[14] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, and D. Walker. TALx86: A Realistic Typed Assembly Language. In *WCSSS*, 1999.

[15] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML Compiler: Performance and Safety Through Types. In *WCSSS*, 1996.

[16] G. C. Necula. Proof-Carrying Code. In *24th POPL*, pages 106–119, 1997.

[17] G. C. Necula. Translation Validation for an Optimizing Compiler. In *PLDI'00*. ACM Press, 2000.

[18] G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *PLDI'98*. ACM Press, 1998.

[19] A. Pnueli, O. Shtrichman, and M. Siegel. Translation Validation for Synchronous Languages. In *ICALP'98*, pages 235–246. Springer-Verlag, 1998.

[20] X. Rival. Abstract Interpretation-based Certification of Assembly Code. In *4th VMCAI*, New York (USA), 2003.

[21] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *PLDI'96*, pages 181–192. ACM Press, 1996.

[22] H. Xi and R. Harper. A dependently typed assembly language. In *International Conference on Functional Programming*, Florence, Italy, September 2001.

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

[24] L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation Run-Time Validation of Optimized Code. In *Electronic Notes in Theoretical Computer Science*, 2002.

A An STFs-based Semantics of a Subset of C

This appendix gives an STFs-based semantics for the subset of C presented on Fig. 1(a).

We suppose there is a label before each statement. A location is either an integer variable or an array variable. Values are integers (hence, $\mathbb{Z} \subseteq R$): overflows are not taken into account.

The STFs define the semantics of statements; hence they describe for each statement the error cases and the transformation of the store in case no error happens. Therefore, we introduce in the next subsection an operation which inputs a symbolic transfer function δ and builds up a symbolic transfer function δ' such that δ' carries out the same transformation as δ in case the evaluation of a given L-value *lv* (resp. an expression *e*, a condition *c*) succeeds and such that δ' fails if the evaluation of *lv* (resp. *e*, *c*) fails. Then, we give the semantics of statements and blocks.

REMARK 2 (ALTERNATE APPROACH). *A valuable approach consists in modifying the semantics of STFs so as to allow failures inside expressions. In case the evaluation of e or lv fails in the environment* ρ*, then* $[\![\lfloor lv \leftarrow e \rfloor]\!](\rho) = \bot_S$*. This approach proved efficient in practice; yet, it would have made the presentation of the paper more involved, so we restrict to the approach evoked in the previous paragraph.*

A.1 Errors in L-values, Expressions and Conditions

We define three operators $\mathcal{C}_{lv} : \mathsf{Lv} \times \mathbb{T} \to \mathbb{T}$, $\mathcal{C}_e : \mathsf{E} \times \mathbb{T} \to \mathbb{T}$ and $\mathcal{C}_c : \mathsf{C} \times \mathbb{T} \to \mathbb{T}$. Intuitively, $\mathcal{C}_{lv}(lv, \delta)$ tests whether the evaluation of *lv* fails; in case *lv* does not fail, then it carries out the same "action" as δ. The operators $\mathcal{C}_e$ and $\mathcal{C}_c$ behave similarly. They are defined by mutual induction (as usual, *x* denotes a variable; *e* an expression; *c*

a condition; lv an l-value, n an integer and δ an STF):

$$\begin{array}{rcl}
\mathcal{C}_{lv}(x,\delta) & = & \delta \\
\mathcal{C}_{lv}(x[e]) & = & \mathcal{C}_e(e, \lfloor (0 \le e) \wedge (e < n)\ ?\ \delta \mid \square \rfloor) \\
 & & (n \text{ is the size of the array } x) \\
\mathcal{C}_e(n,\delta) & = & \delta \\
\mathcal{C}_e(lv,\delta) & = & \mathcal{C}_{lv}(lv,\delta) \\
\mathcal{C}_e(e_0 \oplus e_1,\delta) & = & \mathcal{C}_e(e_0, \mathcal{C}_e(e_1,\delta)) \quad (\oplus \in \{+,-,\star\}) \\
\mathcal{C}_e(e_0/e_1,\delta) & = & \mathcal{C}_e(e_0, \mathcal{C}_e(e_1, \lfloor e_1 = 0\ ?\ e_0/e_1 \mid \square \rfloor)) \\
\mathcal{C}_c(c,\delta) & = & \delta \quad (c \in \{\textbf{true}, \textbf{false}\}) \\
\mathcal{C}_c(\neg c,\delta) & = & \mathcal{C}_c(c,\delta) \\
\mathcal{C}_c(c_0 \oplus c_1,\delta) & = & \mathcal{C}_c(c_0, \mathcal{C}_c(c_1,\delta)) \quad (\oplus \in \{\wedge,\vee\}) \\
\mathcal{C}_c(e_0 \oplus e_1,\delta) & = & \mathcal{C}_e(e_0, \mathcal{C}_e(e_1,\delta)) \quad (\oplus \in \{<,==\})
\end{array}$$

For instance, if t is an array of length 4, then $\mathcal{C}_e(18/t[i],\delta) = \lfloor (0 \le i) \wedge (i < 4)\ ?\ \lfloor t[i] = 0\ ?\ \delta \mid \square \rfloor \mid \square \rfloor$.

If t is an array of length n, the predicate $\textbf{isindex}(t,i)$ is defined by $\textbf{isindex}(t,i) \Leftrightarrow ((0 \le i) \wedge (i < n))$.

In case overflows would be taken into account and the range for the integers would be $[N_{min}, N_{max}]$, then some of the case above would be modified. For instance, the addition would be handled as follows:

$$\mathcal{C}_e(e_0 + e_1, \delta) = \lfloor (N_{min} \le e_0 + e_1) \wedge (e_0 + e_1 \le N_{max})\ ?\ \delta \mid \square \rfloor$$

A.2 Symbolic Transfer Functions for Statements

We present here the transfer functions for each language construction (the STFs that are not mentioned explicitly are $\square$):

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

$$\delta_{l,l'} = \mathcal{C}_{lv}(lv, \mathcal{C}_e(e, \lfloor lv \leftarrow e \rfloor))$$

- Case of a conditional $l : \textbf{if}(c)\,\{l_t : B_t; l_t'\}\,\textbf{else}\,\{l_f : B_f; l_f'\}; l' : ...$:

$$\begin{array}{l}
\delta_{l,l_t} = \mathcal{C}_c(c, \lfloor c\ ?\ \mathfrak{u} \mid \square \rfloor) \\
\delta_{l,l_f} = \mathcal{C}_c(c, \lfloor c\ ?\ \square \mid \mathfrak{u} \rfloor) \\
\delta_{l_t',l'} = \delta_{l_f',l'} = \mathfrak{u}
\end{array}$$

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

$$\begin{array}{rl}
\delta_{l,l_b} = \delta_{l_b',l_b} & = \mathcal{C}_c(c, \lfloor c\ ?\ \mathfrak{u} \mid \square \rfloor) \\
\delta_{l,l'} = \delta_{l_b',l'} & = \mathcal{C}_c(c, \lfloor c\ ?\ \square \mid \mathfrak{u} \rfloor)
\end{array}$$

For instance, if t is an array of length 4 and x is an integer variable, then the assignment $l : x := 18/t[i]; l' : ...$ is represented by the STF:

$$\delta_{l,l'} = \lfloor (0 \le i) \wedge (i < 4)\ ?\ \lfloor t[i] \neq 0\ ?\ \lfloor x \leftarrow 18/t[i] \rfloor \mid \square \rfloor \mid \square \rfloor$$

B STFs-based Semantics of an Assembly Language

We keep the notations of Fig. 1(b) In the following, if x is an integer, then:

- $\underline{x}$ denotes the address of x;
- $\textbf{isaddr}(x)$ is the predicate "x is a valid address in the input store";
- $\textbf{content}(x)$ denotes the content of the memory cell of address x in the input store (addresses are integers).

The values are integers and comparison values. A comparison value is either LT (which means "less than"), or EQ ("equal") and GT ("greater than"). We do not take overflows into account. Hence, $\mathbb{Z} \cup \{\text{LT}, \text{EQ}, \text{GT}\} \subseteq R$.

We list the STFs defining the semantics of the assembly language presented on Fig. 1(b) (the STFs that are not presented explicitly are $\square$):

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

$$\delta_{l,l'} = \lfloor r_0 \leftarrow n \rfloor$$

- the "load" instruction $l : \texttt{load}\ r_0, \underline{x}(v); l' : ...$ 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):

$$\delta_{l,l'} = \lfloor \textbf{isaddr}(\underline{x} + v)\ ?\ \lfloor r_0 \leftarrow \textbf{content}(\underline{x} + v) \rfloor \mid \square \rfloor$$

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

$$\delta_{l,l'} = \lfloor \textbf{isaddr}(\underline{x} + v)\ ?\ \lfloor \underline{x} + v \leftarrow r_0 \rfloor \mid \square \rfloor$$

- the "compare" instruction $l : \texttt{cmp}\ r_0, r_1; l' : ...$ compares the content v_0 and v_1 of the registers r_0 and 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:

$$\delta_{l,l'} = \lfloor r_0 < r_1\ ?\ \lfloor cr \leftarrow LT \rfloor \mid \lfloor r_0 = r_1\ ?\ \lfloor cr \leftarrow EQ \rfloor \mid \lfloor cr \leftarrow GT \rfloor \rfloor \rfloor$$

- the "conditional branching" instruction $l : \texttt{bc}(<)\ l''; l' : ...$ branches to l'' or to the next instruction depending on the value stored in the condition register:

$$\begin{array}{l}
\delta_{l,l'} = \lfloor cr = LT\ ?\ \square \mid \mathfrak{u} \rfloor \\
\delta_{l,l''} = \lfloor cr = LT\ ?\ \mathfrak{u} \mid \square \rfloor
\end{array}$$

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

$$\begin{array}{l}
\delta_{l,l''} = \mathfrak{u} \\
\delta_{l,l'} = \square
\end{array}$$

- the "addition" instruction $l : \texttt{add}\ r_0, r_1, v; l' : ...$ adds the content of the register r_1 and the value v (v is either the content of a register r_2 or an integer constant n) and stores the result in the register r_0 (and the same for the "subtract" and the "multiply" instructions):

$$\delta_{l,l'} = \lfloor r_0 \leftarrow r_1 + v \rfloor$$

In case the valid integer values would be $[N_{min}, N_{max}] \cap \mathbb{Z}$ and if overflows were taken into account, then the definition of the transfer function would be:

$$\delta_{l,l'} = \lfloor (N_{min} \le r_1 + v) \wedge (r_1 + v \le N_{max})\ ?\ \lfloor r_0 \leftarrow r_1 + v \rfloor \mid \square \rfloor$$

- the "division"instruction $l : \texttt{div}\ r_0, r_1, v; l' : ...$ fails if the divisor is equal to 0:

$$\delta_{l,l'} = \lfloor v \neq 0\ ?\ \lfloor r_0 \leftarrow r_1/v \rfloor \mid \square \rfloor$$