Representing Object Code

Marco Benini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39/41 — 20135, Milano, Italy
benini@dsi.unimi.it

Abstract. In this paper, a logical representation of object code programs is presented. The coding is particularly well-suited for mechanization, and it enjoys interesting properties with respect to some relevant approaches to program synthesis, program derivation and formal verification [FD93,LO94,KLO96,FLO97a,LO98]. The paper describes both the representation with its properties, and a tool which permits to translate object programs for the MC68000 microprocessor into the formalism of the ISABELLE logical framework.

1 Introduction

In a verification system it is very important to be able to mechanically represent programs in the formal language used to reason on them. In fact, if the representation of code is left to the human verifier, there is no guarantee that the represented program and the original code describe the same computational process. In this paper, object code programs are considered because they constitute an important area in the application of formal methods as remarked, e.g., in [Yu93,BKN98].

A logical representation of an object code program should meet three conditions with respect to the formal verification task:

- it must be *faithful*, that is, it has to admit a standard interpretation which maps to the same computational process as the original code;
- it must be *meaningful*, that is, it has to allow the full exploitation of the power of the formal system used to reason about programs;
- it must be *intelligible*, so that the relation between the original code and its representation is as plain as possible.

The first point reduces to say that the represented code is equivalent to the original program; to meet this goal, usually, one requires that the representation has to be as close as possible to the original program, under the standard interpretation. Hence, one uses the third point, intelligibility, in addition to the formal semantics of the logical system, to fulfill the requirement for a faithful representation. In this case, simplicity of representation is not a fault, but, on the contrary, it is a benefit because it highly enhances the confidence in the formal proofs.

The second requirement, significance, has the same importance as the others, in fact, a *poor* representation of object code does not permit to use the formal system to its full power, with the result that a correctness proof is harder to obtain, and it appears longer and trickier than necessary.

The proposed representation is very simple, and very close to the description of the MC68000 assembly language one can find in the data book of the microprocessor, thus clearly meeting the requirements of intelligibility and faithfulness. Moreover, the proposed representation admits a standard schema for correctness proofs, which inductively unfolds the possible computations of the program; this fact shows that the representation is meaningful. Furthermore, the proposed representation assumes a distinguished relevance when coupled with a constructive formal system.

In this respect, it is worth noticing that most correctness proofs have been developed using higher-order classical logic, e.g., [Cam88,BG90,Cho94,BP97], or using first-order classical theories extended by computational logics, see, e.g., [Man69,Dij75,BM85,Yu93]. However, as one can easily check by looking at the previously mentioned proofs, most formal verification tasks are developed according to constructive guidelines: as a matter of facts, most correctness proofs do not focus on showing that a program cannot produce but the intended behavior, but, on the contrary, they prove that a program *computes* a specification.

The idea is to assign a meaning to specifications as requests for computations. Nevertheless, this view is reductive: the meaning of a specification may also be prescriptive, that is, the specification acts like a constraint the program is not allowed to violate. These different meanings of specifications are referred to with the words *liveness* for the computational reading, and *safety* for the constraint-oriented view, as customary in the field.

In a constructive approach, the computational reading of a formula is defined by induction on the structure of the formula:

- every atomic formula represents a request for an elementary computation;
- the formula $A \wedge B$ represents a request for a computation which satisfies both A and B, thus, it represents a request for both the computations represented by A and B;
- the formula $A \vee B$ represents a request for a computation which decides between A and B, that is, a computation for $A \vee B$ is either a computation for A or a computation for B;
- the formula $A \to B$ represents a request for a computation which translates any computation for A into a computation for B;
- the formula $\forall x.\, A(x)$ represents a request for a computation of $A(t)$ with an arbitrary input t;
- the formula $\exists x.\, A(x)$ represents a request for a computation of an output t which satisfies the computation requirement represented by $A(t)$.

A definition of the negation case depends on the particular constructive logic: in the intuitionistic case, a specification of the form $\neg A$ requires that the constraint

A is not satisfied by the program, in other words, negation is the way to specify safety properties[1].

The *computational reading* of formulas can be developed only in a constructive framework, because it relies on the ability to formally derive *witnesses* for disjunctive and existential specifications. For a detailed formal treatment of the computational readings for formulas and proofs the reader is referred to [MO81,MMO88,MMO91,LO94,KLO96,FLO97a,FLO97b,FFM99,Ben00].

In a very natural way, liveness properties are modeled as theorems of a constructive logical system, since their proofs require the unfolding of the computations performed by the examined program. On the contrary, safety properties are best modeled in a classical environment, and their proof often requires the use of the *tertium non datur* principle.

Therefore, the ultimate purpose of this paper is to describe a logical representation for object code programs which is compatible with a constructive approach to formal verification, while remaining efficient in a classical approach.

The requirement of efficiency justifies an analysis of compression techniques, which, discarding information which may appear as non relevant to the development of a correctness proof, make the representation more compact, more understandable, and thus, more manageable by a semiautomatic theorem prover.

2 Translating Object Code into Logic

The notion of object code is slightly ambiguous; in fact, it may be interpreted in four different ways:

1. The object code of a program is the content of the computer memory when the operative system passes it the control;
2. The object code of a program is the content of an executable file;
3. The object code of a program is the output of a compiler, in other words, the content of an *object* (`.o`) file;
4. The object code is the symbolic representation of the output of a compiler, i.e., it is an assembly program.

All these possibilities are right, to some extent, and all of them are supported by the translation tool, OCT (Object Code Translator).

The OCT tool is divided into two parts, the *preprocessor* and the *translation procedure*; the former transforms an executable file or an object file into an assembly program; the latter takes an assembly program as input and produces a logical theory suitable for reasoning with ISABELLE [Pau94].

The preprocessor reconstructs an assembly program where every address is *resolved*, that is, the assembly code is allocated in memory from a given address.

[1] More precisely, one may express a safety property A by means of its double negation, $\neg\neg A$. To maintain compatibility with the classical interpretation of logical symbols, it is necessary to work in Kuroda logic [Gab81,AFMM96,MMO97], that is, **IL** (intuitionistic first-order logic) plus the Kuroda axiom, $(\forall x.\, \neg\neg A(x)) \rightarrow \neg\neg \forall x.\, A(x)$.

Thus, the output of the preprocessor is equivalent to the symbolic (assembly) representation of the content of the memory when the program will be executed.

The translation procedure takes as input an assembly source code where no macros are present and where every address is resolved, and it translates this code into a logical representation.

The target assembly language is the one of the MC68000 microprocessor; the main reason behind the choice of this particular architecture is that several case studies for formal verification problems have been developed starting from the MC68000 microprocessor; in particular, the translation tool benefits from the good work in [Yu93], where many functions from the standard `libc` library have been proven correct, providing a consistent set of test cases.

The translation algorithm operates on the language of first-order intuitionistic logic plus the theory of modular arithmetic where the types `byte`, `word` and `longword` are integers modulo 2^8, integers modulo 2^{16} and integers modulo 2^{32}, respectively. Modular arithmetic is available in the ISABELLE framework, and there are reasoners which can efficiently deal with it, see [Sho79,CLS96,BNP98].

The output of the translation procedure is an ISABELLE theory containing a series of axioms, one per instruction, encoding the program. This theory file inherits the necessary type declarations as well as the constants representing registers and memory from the theory of the microprocessor.

The theory of the microprocessor has three roles:

- it provides the minimal set of instruments to reason about object code programs;
- it declares the types which are needed to represent the code;
- it declares the constants which constitute the world the microprocessor operates on.

The set of instruments is given by the logical system, the theory of identity, the set of provers, such as the *Simplifier* [Pau97] which allows equational reasoning, the *Computer Arithmetic Toolkit* [BNP98], to deal with modular arithmetic, and the *Classical Reasoner* [Pau97], to cope with purely logical problems.

The types `byte`, `word` and `longword` are specializations of modular numbers. In practice, both signed and unsigned numbers are used. Their coding is declared in the modular arithmetic package. Hence, the microprocessor theory declares three versions for every type; for instance, it defines pure bytes, denoted by the type `byte`, which is the set of integer numbers quotiented by the relation (mod 2^8), signed bytes, denoted by `sbyte` and representing the range of numbers from -128 to 127, and unsigned bytes, denoted by `ubyte` and representing the range of numbers from 0 to 255.

The type `time`, following the fact that the microprocessor clock is discrete, is equivalent to `Int`, that is, time is modeled by integer numbers.

In the microprocessor theory, the constants for memory and registers are declared. Specifically, the MC68000 microprocessor[2] provides sixteen 32-bit user registers, eight of them (the d registers) being data registers, the others (denoted

[2] The details of the MC68000 architecture can be found in [Mot89].

by a) being address registers. The program counter register is indicated with pc. Since registers change their values over time, they have been modeled as functions from time to values:

$$\begin{array}{ll} \mathrm{d}_i : \texttt{time} \to \texttt{slongword} & , 0 \leq i \leq 7 \\ \mathrm{a}_i : \texttt{time} \to \texttt{slongword} & , 0 \leq i \leq 7 \\ \mathrm{pc} : \texttt{time} \to \texttt{ulongword} & \end{array}$$

A particular case is the status register which is modeled by a set of functions, one for each flag in the register:

$$\begin{array}{lll} \texttt{Zflag}: \texttt{time} \to \texttt{bool} & (* \text{ zero} & *) \\ \texttt{Nflag}: \texttt{time} \to \texttt{bool} & (* \text{ negative} & *) \\ \texttt{Cflag}: \texttt{time} \to \texttt{bool} & (* \text{ carry} & *) \\ \texttt{Vflag}: \texttt{time} \to \texttt{bool} & (* \text{ overflow} & *) \\ \texttt{Xflag}: \texttt{time} \to \texttt{bool} & (* \text{ extension} & *) \end{array}$$

The memory is represented as a function from addresses and times to values:

$$\texttt{memory}: \texttt{ulongword} \times \texttt{time} \to \texttt{byte}$$

The general format of the logical representation of an instructions I is

$$\forall\, t: \texttt{time}.\, \mathrm{pc}(t) = A \to B \wedge C$$

where A is the absolute address of the instruction I, B specifies the value of the program counter at time $t+1$, and C specifies the value of every register, flag and memory cell at time $t+1$, depending on the instruction operands, the status of memory at time t, and the values of registers and flags at time t.

The format of the B part can be either

$$\mathrm{pc}(t+1) = H(\mathrm{pc}(t))$$

or

$$(f(t) \to \mathrm{pc}(t+1) = H_1(\mathrm{pc}(t))) \wedge (\neg f(t) \to \mathrm{pc}(t+1) = H_2(\mathrm{pc}(t)))$$

where H, H_1 and H_2 are arithmetical expressions depending on the current value of the program counter and calculating the address of the next instruction to execute; $f(t)$ is a formula, depending on the time t, and usually, it is a literal representing a flag, but, in general it may be a conjunction of (negations of) flag predicates.

For example: the instruction

$$64: \texttt{MOVE}\ \#1, \mathrm{d}_0$$

which puts the value 1 into the d_0 register[3], is translated into

$$\begin{aligned}\forall t.\, \mathrm{pc}(t) = 64 \rightarrow\ & \mathrm{pc}(t+1) = \mathrm{pc}(t) + 2 \wedge \\ & \wedge\ \mathrm{d}_0(t+1) = 1 \wedge \\ & \wedge\ \mathrm{d}_1(t+1) = \mathrm{d}_1(t) \wedge \ldots \wedge \mathrm{d}_7(t+1) = \mathrm{d}_7(t) \wedge \\ & \wedge\ \mathrm{a}_0(t+1) = \mathrm{a}_0(t) \wedge \ldots \wedge \mathrm{a}_7(t+1) = \mathrm{a}_7(t) \wedge \\ & \wedge\ \neg \mathtt{Vflag}(t+1) \wedge \neg \mathtt{Cflag}(t+1) \wedge \neg \mathtt{Zflag}(t+1) \wedge \\ & \wedge\ \neg \mathtt{Nflag}(t+1) \wedge \neg \mathtt{Xflag}(t+1) \wedge \\ & \wedge\ \forall a.\, \mathtt{memory}(a, t+1) = \mathtt{memory}(a, t)\ .\end{aligned}$$

Also, the instruction

$$72 : \mathtt{BEQ}\ 8$$

which represents a conditional branch 8 positions forward if the zero flag is set, is translated into

$$\begin{aligned}\forall t.\, \mathrm{pc}(t) = 72 \rightarrow\ & (\mathtt{Zflag}(t) \rightarrow \mathrm{pc}(t+1) = \mathrm{pc}(t) + 8) \wedge \\ & \wedge\ (\neg \mathtt{Zflag}(t) \rightarrow \mathrm{pc}(t+1) = \mathrm{pc}(t) + 2) \wedge \\ & \wedge\ \mathrm{d}_0(t+1) = \mathrm{d}_0(t) \wedge \ldots \wedge \mathrm{d}_7(t+1) = \mathrm{d}_7(t) \wedge \\ & \wedge\ \mathrm{a}_0(t+1) = \mathrm{a}_0(t) \wedge \ldots \wedge \mathrm{a}_7(t+1) = \mathrm{a}_7(t) \wedge \\ & \wedge\ (\mathtt{Vflag}(t+1) \leftrightarrow \mathtt{Vflag}(t)) \wedge \\ & \wedge\ (\mathtt{Zflag}(t+1) \leftrightarrow \mathtt{Zflag}(t)) \wedge \\ & \wedge\ (\mathtt{Nflag}(t+1) \leftrightarrow \mathtt{Nflag}(t)) \wedge \\ & \wedge\ (\mathtt{Cflag}(t+1) \leftrightarrow \mathtt{Cflag}(t)) \wedge \\ & \wedge\ (\mathtt{Xflag}(t+1) \leftrightarrow \mathtt{Xflag}(t)) \wedge \\ & \wedge\ \forall a.\, \mathtt{memory}(a, t+1) = \mathtt{memory}(a, t)\ .\end{aligned}$$

Some remarks on the proposed representation are needed:

- The simplicity of the representation makes evident its correctness, since it is very adherent to the description found in the data book.
- The preprocessor takes care of eliminating the dependency on the system architecture. Then the translation procedure transforms a symbolic equivalent of the memory image of the program into a logical representation which is faithful. Thus, the result is really equivalent to what will be executed.
- The theory of the microprocessor and the output of the translation procedure are Harrop theories.

An important point, anticipated in the introduction, is that the representation naturally imposes a structure on correctness proofs. In fact, in order to prove that the program P has the property ϕ, a proof of ϕ which unfolds the possible computations of P is required. The general format of that proof is

$$\begin{array}{c} \Gamma, R \\ \vdots \\ \phi \end{array},$$

[3] The set of instructions of the MC68000 microprocessor is documented in [Mot89].

where R is the representation of P and Γ is a set of assumptions specifying, at least, the initial state before the execution of P.

The proof $\begin{array}{c}\Gamma, R\\ \vdots\\ \phi\end{array}$ has a canonical form, since every step in the computation of P can be simulated by a proper application of inference rules; for example, if $\mathrm{pc}(t_0) = n$ and n is a location belonging to the program P, then the execution of the instruction of P at address n is simulated by the following proof schema

$$\dfrac{\dfrac{}{\forall\, t.\, \mathrm{pc}(t) = n \to B(t) \wedge C(t)}}{\dfrac{\mathrm{pc}(t_0) = n \quad \mathrm{pc}(t_0) = n \to B(t_0) \wedge C(t_0)}{B(t_0) \wedge C(t_0)}}$$

where $B(t_0)$ gives the possible values of the program counter at time $t_0 + 1$, and $C(t_0)$ computes the values of registers, flags and memory at time $t_0 + 1$.

As soon as no loops are involved, a combination of instances of the preceding proof schema and applications of the substitution rule really unfolds any possible computation of the program.

When a loop is present in the program, necessarily there is a branching instruction which assigns a value n to the program counter such that the instruction located at n has already been executed. In this case, when composing instances of the preceding proof schema, it always appears a schema of the form

$$\begin{array}{c}\Gamma, R, \mathrm{pc}(t) = n\\ \vdots\\ \mathrm{pc}(t + k) = n\end{array}$$

which naturally suggests to use an induction principle. In the case of structured (non-interleaving) cycles, a convenient choice is the *bounded chain principle*[4]:

$$\dfrac{\dfrac{}{\begin{array}{c}[p \leq b], [P(p)]\\ \vdots\end{array}}}{\exists\, x.\, x \leq b \wedge P(x) \quad B \vee (\exists\, y.\, p < y \leq b \wedge P(y))}$$
$$B$$

which is applied instantiating $P(x)$ to $\mathrm{pc}(x) = n$, and B to $\exists\, t.\, \mathrm{pc}(t) = m$, where m is the location reached when the loop finishes its execution. The bound b has to be guessed, and corresponds to an upper bound for the computational complexity of the loop.

It is possible to mechanically generate the proof schema which inductively unfolds every possible computation of a program; however, the details of this construction, whose pieces have been sketched above, and the proof of its adherence to the microprocessor's semantics are too complex to be presented here.

Essential to remark is the fact that a correctness proof schema can be generated for a program either in a constructive logical system, or in a classical

[4] This induction schema formalizes a specialization of the *descending chain principle* [MO81,MMO88,MMO91,Fer97,Ben00], and is valid in every discrete ordering.

environment. So, in both cases the representation is really meaningful, providing a strong guideline in the development of correctness proofs.

The importance of being an Harrop theory cannot be appreciated without introducing some details on constructive systems. A formal system is said to be *uniformly constructive* [Fer97] when

- if Π is a proof of $A \vee B$, then either there is a proof Π' of A, or there is a proof Π' of B, and Π' is an instance of a combination of subproofs of Π;
- if Π is a proof of $\exists\, x.\, A(x)$, then there is a proof Π' of $A(t)$, and Π' is an instance of a combination of subproofs of Π.

When a proof Π' is required to be an instance of a combination of subproofs of another proof Π, it amounts to demand that the conclusion D of Π' is implicitly proven by Π, or, in other words, that D is in the *truth content* of Π. Actually, one can prove that the truth content of a proof can be algorithmically generated, see [MO81,Fer97,Ben00] for details.

A well-known fact is that, when a set of Harrop axioms is added to an uniformly constructive formal system, the result is another uniformly constructive system, see, e.g., [ST96,Fer97,FFM99], so the discussed representation preserves applicability of the instruments that make use of the truth content of a correctness proof to analyze the corresponding program [Ben00].

The importance of constructive systems in formal verification appears also from the fact that, in an uniformly constructive system, one can assign a *computational meaning* to specifications [MMO88,MMO91,FLO97b,Ben00], as briefly illustrated in the introduction. An important consequence of this fact is related to the possibility to extract information from correctness proofs by generating the associated truth content, and to ensure that the extracted information is enough to symbolically compute the program on an input [MO81].

Henceforth, the proposed representation takes a deeper meaning in a constructive framework, where it admits a computational reading which formally proves that the representation is faithful.

3 Compressing the Representation

The representation of object code as described in the previous section is satisfactory for the purposes illustrated in the introduction, but it suffers from being quite redundant.

In fact, most of the information contained in the formula which encodes an instruction, proves to be useless in practice. For example, considering the fragment

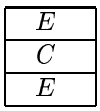
$$\begin{array}{l} 40\colon\ \texttt{MOVE}\ \#7, \mathrm{d}_0 \\ 42\colon\ \texttt{ADD}\ \#1, \mathrm{d}_0 \\ 44\colon\ \texttt{MOVE}\ \mathrm{d}_0, (\mathrm{a}_1) \end{array}$$

when a correctness proof is developed, in most cases, the fragment encoding in a logical form can be reduced to

$$\forall\, t.\, \mathrm{pc}(t) = 40 \rightarrow \mathrm{pc}(t+3) = 46 \wedge \mathrm{d}_0(t+3) = 8 \wedge \texttt{memory}(\mathrm{a}_1(t), t+3) = 8$$

In this section some techniques are presented which can be used to mechanically compress the logical representation.

The *enveloping* technique is based on the fact that compilers produce object code with a peculiar structure; in particular, a procedure is compiled in a way which can be represented as



the C part is the code which implements the procedure, while the E part, the *envelope*, takes care of retrieving the parameters and returning the result.

For example, the following function in the C language:

```
int f( int x )
{
    return (x + 1) ;
}
```

is compiled by the `gcc` compiler into the following object code

```
 0: LINK #0, a6
 2: MOVE 0(a7), d0
 5: ADD #1, d0
 7: MOVE d0, -8(a7)
10: UNLK a6
12: RTS
```

the parts which constitute the body of the function and its envelope are marked.

The envelope compression technique takes apart the representation for the envelope, proves that it correctly passes through parameters, and proves that it correctly returns the result; these proofs are routine and they can be efficiently mechanized. In such a way, the human verifier has only to prove that the body of the procedure is correct.

The drawback of the enveloping compression technique is that it requires that the object code is organized according the envelope pattern, which is not always the case for human-produced or highly optimized code.

Very important to remark is the fact that this compression technique does not discard information, thus, when applicable, it is safe.

The analysis of the flow of control of a program is probably the most important technique for compressing the logical representation. It is based on the grouping of sequential blocks of instructions.

The algorithm which performs this kind of compression is complex because of the amount of details, but its main structure can be described as a transformation on graphs: given an assembly program P, a graph is constructed whose nodes are

the instructions of P and whose directed edges are drawn from an instruction I to any instruction J which may be executed just after I. The compression of sequential blocks can be formulated as a transformation on this graph which collapses two nodes A and B if there is an edge from A to B, and no edges of the form A to C, or C to B, for any node C.

For example, the fragment of code at the beginning of this section, after the compression, is represented by the formula

$$\begin{aligned}\forall t.\,\mathrm{pc}(t) = 40 \rightarrow\ & \mathrm{pc}(t+3) = \mathrm{pc}(t) + 6 \wedge \\ & \wedge\, \mathrm{d}_0(t+3) = 8 \wedge \\ & \wedge\, \mathrm{d}_1(t+3) = \mathrm{d}_1(t) \wedge \ldots \wedge \mathrm{d}_7(t+3) = \mathrm{d}_7(t) \wedge \\ & \wedge\, \mathrm{a}_0(t+3) = \mathrm{a}_0(t) \wedge \ldots \wedge \mathrm{a}_7(t+3) = \mathrm{a}_7(t) \wedge \\ & \wedge\, \neg \mathtt{Vflag}(t+3) \wedge \neg \mathtt{Zflag}(t+3) \wedge \\ & \wedge\, \neg \mathtt{Nflag}(t+3) \wedge \neg \mathtt{Cflag}(t+3) \wedge \neg \mathtt{Xflag}(t+3) \wedge \\ & \wedge\, \mathtt{memory}(\mathrm{a}_1(t), t+3) = 8 \wedge \\ & \wedge\, \forall x.\, x\neg = \mathrm{a}_1(t) \rightarrow \mathtt{memory}(x, t+3) = \mathtt{memory}(x, t)\ .\end{aligned}$$

An important point about the compression of sequential code is the fact that the compressed representation is again an Harrop theory, thus preserving uniform constructivity and, more general, the benefits of the chosen representation.

Moreover, the algorithm which generates the compressed representation can be easily employed to compute a proof schema for the program, starting from the general correctness proof schema and reducing it to use just the compressed representation.

However, although the compression of sequential code does not reduce the amount of information about what the original program computes, it destroys the information on the order in which sequential computations are performed. Occasionally this matters, especially when dealing with safety properties. Nevertheless, if ϕ is a formula which does not contain explicit references to values of time not appearing in the compressed representation, and, moreover, if ϕ does not contain existentially quantified subformulas over time variables, then the following theorem holds in **CL** (classical first-order logic):

Theorem 1. *Let R be the theory which contains the representation of a program P, let R_c be the theory containing the representation of P where sequential blocks are compressed, and let ϕ be a formula as above, then, if* $\mathbf{CL}, R, \Gamma \vdash \phi$, *then* $\mathbf{CL}, R_c, \Gamma \vdash \phi$.

This theorem constitutes a preservation result for a large subclass of liveness specifications, remarking the importance of the compression algorithm. The proof of the theorem is laborious, so it is omitted for the sake of brevity; it is based on the subformula property of the normalization theorem for **CL** [ST96].

The previous theorem can be strengthened when working in a constructive system which contains the intuitionistic logic plus the Kuroda principle: in that case, in fact, the formula ϕ is required not to contain explicit references to values of time not appearing in the compressed representation, and, ϕ is required not to contain an occurrence of an existentially quantified subformula $\exists t.\, A(t)$ over

a time variable t, *unless* the witness $A(t)$ appearing[5] in the truth content of $\mathbf{IL}, R, \Gamma \vdash \phi$ meets the conditions on ϕ, too. In practice, most of the times, it is evident form the program that a witness $A(t)$ exists and that t is in the time domain of the compressed representation.

The analysis of the flow of control of a program naturally gives raise to another compression technique: the idea is to maintain in the representation just what is or was or will be changed inside the program. In practice, the equalities stating that a register (memory cell, flag) retains the same value since it is neither read, nor written by the program, are deleted from the representation.

The algorithm which performs the compression of equalities is a variant of the algorithm which compresses sequential blocks. In fact, it is modeled by a transformation on a labelled version of the graph of a program as previously defined. Being G the graph associated to the program P; every node N is labelled with the set C_N of registers, flags and memory cells that are changed by the execution of the instruction the node represents; moreover, every node N of G is labelled with the set R_N of registers, flags and memory cells which are read by the instruction in N, that is, the registers, flags and cells which are on the right-hand side of equalities whose left-hand side is in C_N. The transformation that performs the compression operates as follows: let N and M be two nodes connected by an arc from N to M, the labels of N are updated as $C'_N = C_N \cup R_M$ and $R'_N = R_N \cup (R_M \setminus C_N)$. The least fixed point of this transformation produces a graph G^* such that, for every node N in G^*, the label C_N contains exactly the left-hand side of the equalities which must be retained in the compressed representation of the instruction associated with N.

Furthermore, the two compression techniques derived from the analysis of the flow of control may be combined in a single compression algorithm. In the example from the beginning of this section, the combined algorithm produces

$$\forall\, t.\, \mathrm{pc}(t) = 40 \rightarrow \mathrm{pc}(t+3) = 46 \wedge \mathrm{d}_0(t+3) = 8 \wedge \mathtt{memory}(\mathrm{a}_1(t), t+3) = 8$$

for the whole sequential block.

As before, the result of the compression of equalities, as well as, the result of the combined algorithm, is an Harrop theory, thus preserving significance of the representation. Moreover, as before, a correctness proof schema for the program can be generated by the compression algorithm.

On the other hand, the compression of equalities discards more information than the compression of sequential blocks, and, as a result, the previous preservation theorem does not hold anymore. Still, there is a characterization of formulas whose provability is preserved under the compression of equalities. However, the combined compression algorithm is usually preferred, hence a preservation result for the single compression technique is not shown, in favor of a characterization of the formulas preserved by the combined transformation.

Let ϕ be a formula such that:

[5] Being constructive, the system guarantees the appearance of such a witness in the truth content of the proof, see [MO81].

1. it does not contain explicit references to values of time non appearing in the compressed representation;
2. it does not contain occurrences of timed registers, timed flags and timed memory locations non appearing in the compressed representation;
3. it does not contain existentially quantified subformulas over a time variable;

then, the following preservation theorem holds

Theorem 2. *Let R be the theory which contains the representation of a program P, let R_c be the theory containing the representation of P compressed according to the combined algorithm, and let ϕ be a formula as above, then, if* $\mathbf{CL}, R, \Gamma \vdash \phi$, *then* $\mathbf{CL}, R_c, \Gamma \vdash \phi$.

The proof follows the same pattern as the previous preservation result, thus, it heavily uses the subformula property and the normalization theorem for **CL**. As in the previous case, the conditions on ϕ can be relaxed in Kuroda logic, allowing existentially quantified subformulas on time, whose witnesses are in the time domain to the compressed representation.

It is evident that the class of formulas whose provability is preserved by the combined algorithm is a proper subclass of those preserved by the compression of sequential blocks. In this respect, the combined algorithm is stronger, as remarked before.

On the other hand, it should be clear that the class of formulas preserved by the combined algorithm is *natural*, since the requirements it has to satisfy is, intuitively, that the formulas do not contain references to objects (registers, flags, memory cells and times) which have been discarded in the compression process.

An apparently interesting variation on the previous combined algorithm is given by considering a specification S of the form $\exists t.\, \mathrm{pc}(t) = x \wedge A$, which constitutes an usual pattern for liveness properties, and compressing the code with respect to S, that is, to keep in the representation only the objects (registers, flags and memory cells) which are referred in S.

The algorithm which computes a compressed representation for a program P, according to this requirement is a variant of the algorithm which performs the compression of equalities: being G the labelled graph for the program P constructed as above, being S the specification, which, by assumption, has the form $\exists t.\, \mathrm{pc}(t) = x \wedge A$, and calling N the node corresponding to the instruction at the address x, a new labelled graph G' is constructed, where G' is equal to G except for the node N, where the label C_N is substituted with the set of all registers, flags and memory cells occurring in S. The result of the compression algorithm is the result on running the combined compression procedure on G'.

It is evident that the result is a representation which is the minimal one computing just the values mentioned in the specification. But the resulting representation may be (and it is often the case) too poor to allow to prove the specification itself.

In fact, no *natural* preservation result is known at the moment being, and there is the strong belief that no reasonable preservation theorem can be proven. In fact, characterizing the class of formulas whose provability is preserved by this kind of compression procedure appears to be hopeless. However, it is possible to describe some restricted classes of formulas which enjoy the mentioned preservation property, but none of them is significantly large, or representative of a wide class of specifications.

However, although in general the previous claim holds, in a constructive setting some preliminary results give an hope. Precisely, a compression algorithm which operates as above, but takes additional information on unsuccessful proof attempts in account is currently being studied. This approach uses information extraction algorithms from a partial proof, the failed attempt, to analyze the lack of information which prevented the completion of the correctness proof. The lacking information, which is part of the full representation of the program, is added to the set of objects the compression algorithm has to retain, and the partial correctness proof is redone, up to the point where it failed; since new information is available, new inference steps can be performed, and more chances for a success are gained. Unfortunately, no definitive result on what is preserved by this approach is currently available.

4 Conclusions

In this paper a simple way to represent object code programs has been introduced, and it has been made clear that it is suitable for mechanization. Moreover, the chosen representation can be compressed with various techniques, without affecting the provability of wide classes of specifications.

The novelty of the contribution lies in three points:

1. A general proof schema for correctness proofs can be generated along with the representation. It corresponds to a proof by induction on the paths of computation of the represented program.
2. The provability in both classical and Kuroda logic of a wide class of specifications is not affected by two important compressions of the representation of a program.
3. In a proper constructive framework, the addition of the theory of the microprocessor and the theory containing the program representation generates an enlarged logical system which is still constructive. Moreover, in a constructive system the preservation results on the compression algorithms can be slightly extended.

About the last point, it is worth remarking that, although constructive systems are not usually employed in formal verification, most correctness proofs have a constructive flavor, because this is what a human reader asks for to be convinced by the proof itself. More important, proving techniques based on constructive methods allow deeper kinds of analysis of the resulting correctness proofs, as discussed in [Ben00]. Thus the presented work is also a first effort to introduce these novel techniques to the formal verification community.

References

[AFMM96] A. Avellone, C. Fiorentini, P. Mantovani, and P. Miglioli. On maximal intermediate predicate constructive logics. *Studia Logica*, 57:373–408, 1996.

[Ben00] M. Benini. *Verification and Analysis of Programs in a Constructive Environment*. PhD thesis, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, January 2000.

[BG90] G. Birtwistle and B. Graham. Verifying SECD in HOL. In J. Staunstrup, editor, *Proceedings of the IFIP TC10/WG10.5 Summer School on Formal Methods for VLSI Design*. North Holland, 1990.

[BKN98] M. Benini, S. Kalvala, and D. Nowotka. Program abstraction in a higher-order logic framework. In J. Grundy and M. Newey, editors, *Proceedings of Theorem Proving in Higher-Order Logic '98 International Conference*, volume 1479 of *Lecture Notes in Computer Science*, pages 33–48. Springer Verlag, 1998.

[BM85] R.S. Boyer and J.S. Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[BNP98] M. Benini, D. Nowotka, and C. Pulley. Computer arithmetic: Logic, calculation and rewriting. In D.M. Gabbay and M. De Rijke, editors, *Frontiers of Combining Systems 2*, Series in Logic and Computation, pages 77–93. Research Studies Press, 1998.

[BP97] G. Bella and L.C. Paulson. Using Isabelle to prove properties of the Kerberos authentication system. In H. Orman and C. Meadows, editors, *Workshop on Design and Formal Verification of Security Protocols*. DIMACS, September 1997.

[Cam88] A.J. Camilleri. *Executing Behavioural Definitions in Higher-Order Logic*. PhD thesis, Cambridge University, February 1988. Technical Report No 140, Computer Laboratory, Cambridge University.

[Cho94] C.T. Chou. Mechanical verification of distributed algorithms in higher-order logic. In T.F. Melham and J. Camilleri, editors, *Higher-Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 158–176. Springer Verlag, September 1994.

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477. Springer Verlag, 1996.

[Dij75] E.W. Dijkstra. Guarded commands, non determinacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–458, 1975.

[FD93] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation*, 15(5–6):775–806, 1993.

[Fer97] M. Ferrari. *Strongly Constructive Formal Systems*. PhD thesis, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1997.

[FFM99] M. Ferrari, C. Fiorentini, and P. Miglioli. Extracting information from intermediate **T**-systems. In *Intuitionistic Modal Logics and Applications*, Trento, Italy, 1999. Federated Logic Conference.

[FLO97a] P. Flener, K.K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In *Proceedings XIIth IEEE International Automated Software Engineering Conference*, pages 153–160, 1997.

[FLO97b] P. Flener, K.K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs, editor, *Proceedings of the 7th International Workshop on Logic-Based Program Synthesis and Transformation*, Lecture Notes in Computer Science. Springer Verlag, 1997.

[Gab81] D.M. Gabbay. *Semantical Investigations in Heyting Intuitionistic Logic*. D. Reidel Publishing Company, Dordrecht, 1981.

[KLO96] C. Kreitz, K.K. Lau, and M. Ornaghi. Formal reasoning about modules, reuse and their correctness. volume 1085 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1996.

[LO94] K.K. Lau and M. Ornaghi. A formal view of specification, deductive synthesis and transformation of logic programs. In Y. Deville, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'93*, Workshops in Computing, pages 10–31. Springer Verlag, 1994.

[LO98] K.K. Lau and M. Ornaghi. Isoinitial models for logic programs: A preliminary study. In J.L. Freire-Nistal, M. Falaschi, and M. Vilares-Ferro, editors, *Proceedings of the 1998 Joint Conference on Declarative Programming*, pages 443–455, 1998.

[Man69] Z. Manna. Properties of programs and the first order predicate calculus. *Journal of the Association for Computing Machinery*, 16(2), 1969.

[MMO88] P. Miglioli, U. Moscato, and M. Ornaghi. Constructive theories with abstract data types for program synthesis. In D. Skordev, editor, *Mathematical Logic and its Applications*, pages 293–302. Plenum Press, 1988.

[MMO91] P. Miglioli, U. Moscato, and M. Ornaghi. Program specification and synthesis in constructive formal systems. In K.K. Lau and T.P. Clement, editors, *Logic Program Synthesis and Transformation, Manchester 1991*, pages 13–26. Springer Verlag, 1991.

[MMO97] P. Miglioli, U. Moscato, and M. Ornaghi. Avoiding duplications in tableau systems for intuitionistic and Kuroda logics. *Logical Journal of the IGPL*, 1(5):145–167, 1997.

[MO81] P. Miglioli and M. Ornaghi. A logically justified model of computation. *Fundamenta Informaticæ*, IV(1,2), 1981.

[Mot89] Motorola Inc., editor. *MC68020 32-bit Microprocessor User's Manual*. Prentice Hall, New Jersey, 1989.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Pau97] L.C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications*, chapter 3. The MIT Press, 1997. Also, Report No 396, Computer Laboratory, Cambridge University.

[Sho79] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the Association for Computing Machinery*, 26(2):351–360, 1979.

[ST96] H. Schwichtenberg and A.S. Troelstra. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[Yu93] Y. Yu. Automated proofs of object code for a widely used microprocessor. Technical Report 114, Digital Equipment Corporation, Systems Research Center, October 1993.