Principales publications

Ce document rassemble mes 5 principales publications, présentées dans l'ordre des chapitres avec leur numéro de référence dans la bibliographie.

(chapitre 3)

[5] Sandrine Blazy, Philippe Facon. *Partial Evaluation for the understanding of Fortran programs.*
Journal of Software Engineering and Knowledge Engineering. Vol. 4, numéro 4, 1994, pp. 535-559.

[1] Sandrine Blazy. *Specifying and Automatically Generating a Specialization Tool for Fortran 90.*
Journal of Automated Software Engineering. Vol. 7, numéro 4. Décembre 2000, pp. 345-376.

(chapitre 5)

[7] Sandrine Blazy, Zaynah Dargaye, Xavier Leroy. *Formal verification of a C compiler front-end.*
14^e "Symposium on Formal Methods" (FM'06), Hamilton, Canada, 23 au 25 août 2006. Lecture Notes in Computer Science (LNCS) n°4085, Springer Verlag, pp.460-475.

(chapitre 6)

[10] Xavier Leroy, Sandrine Blazy. *Formal verification of a C-like memory model and its uses for verifying program transformations.*
Journal of Automated Reasoning. Vol.41, numéro 1, juillet 2008, pp. 1-31.

(chapitre 7)

[11] Andrew W. Appel, Sandrine Blazy. *Separation logic for small-step Cminor.*
20^e conférence internationale "Theorem Proving in Higher Order Logics" (TPHOLs 2007), Kaiserslautern, Allemagne, 10 au 13 septembre 2007. Lecture Notes in Computer Science (LNCS) n°4732, Springer Verlag, pp.5-21.

International Journal of Software Engineering and Knowledge Engineering
Vol. 4 No. 4 (1994) 535–559

PARTIAL EVALUATION FOR THE UNDERSTANDING OF FORTRAN PROGRAMS*

SANDRINE BLAZY† and PHILIPPE FACON
CEDRIC IIE, 18 allée Jean Rostand, 91025 Evry Cedex, France
E-mail: {*blazy, facon*} *@iie cnam fr*

Received 15 October 1993
Revised 20 June 1994
Accepted 20 September 1994

This paper describes a technique and a tool that support partial evaluation of FORTRAN programs, i e., their specialization for specific values of their input variables The authors' aim is to understand old programs, which have become very complex due to numerous extensions. From a given FORTRAN program and these values of its input variables, the tool provides a simplified program, which behaves like the initial program for the specific values This tool mainly uses constant propagation and simplification of alternatives to one of their branches. The tool is specified in terms of inference rules and operates by induction on the FORTRAN abstract syntax These rules are compiled into Prolog by the Centaur/FORTRAN programming environment The completeness and soundness of these rules are proven using rule induction

Keywords: FORTRAN, software maintenance, program understanding, program specialization, partial evaluation, proof of completeness and soundness, Centaur.

1. Introduction

Program understanding is the most expensive phase of the software life-cycle. It is said that 40% of the maintenance effort is spent trying to understand how existing software works [23] All maintenance problems do not require complete program understanding, but each problem requires at least a limited understanding of how the source code works, and how it is related to the external functions of the application. There exists now a wide range of tools to support program understanding [24].

Program slicing is a technique for restricting the behavior of a program to some specified subset of interest. The slice of a program P on a subset V of the variables of P at location i is the set of all the statements that might affect the values of the variables in V at i. This is an executable program that is obtained by data flow

*This paper is an extended version of a paper that received the student paper award at the SEKE'93 conference
†Sandrine Blazy was previously at the EDF Research Center, Clamart, France

analysis. Program slicing can be used to help maintainers understand and debug code [11]

We have developed a complementary technique: reduction of programs for specific values of their input variables. It aims at understanding old programs, which have become very complex due to extensive modifications. For a given FORTRAN program and some form of restriction of its usage (e.g., the knowledge of some specific values of its input variables), the tool provides a simplified program, which behaves like the initial program when used according to the restriction. *This approach is particularly well adapted to programs which have evolved as their application domains increase continuously.*

Partial evaluation is an optimization technique used in compilation to specialize a program for some of its input variables. Partial evaluation of a subject program P with respect to input variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ for the values $x_1 = c_1, \ldots, x_m = c_m$ gives a residual program P', whose input variables are $y_1, \ldots, y_n$, and the executions of $P(c_1, \ldots, c_m, y_1, \ldots, y_n)$ and $P'(y_1, \ldots, y_n)$ produce the same results [19]. Such a program is obtained by replacing variables by their constant values, by propagating constant values, and by simplifying statements; for instance, replacing each alternative whose condition simplifies to a constant value (true or false) by the corresponding branch.

Partial evaluation has been applied to program optimization and compiler generation from interpreters (by partially evaluating the interpreter for a given program) [15]. In this context, previous work has primarily dealt with functional [3] and declarative languages [22]. The structure of the program may be modified (using loop expansion, and subroutine expansion and renaming [9]) in order to optimize the residual code.

As far as imperative languages are concerned, partial evaluation has been used for software reuse improvement by restructuring software components to improve their efficiency [4]. Partial evaluation has been applied to numerical computation to provide performance improvements for a large class of numerical programs, by eliminating data abstractions and procedure calls [5].

Our goal is different. We remove groups of statements that are never used in the given context, but we do not expand statements. This does not change the original structure of the code. We transform general-purpose programs into shorter and easier-to-understand special-purpose programs. This transformational approach aims at improving a given program without disturbing its correctness when used in a given restricted and stable context. However, unlike [17], we do not aim at improving a program according to a performance criterion (e.g., memory), but at improving the readability of programs.

What are the interesting program simplifications in that context? We believe that to remove useless code is always beneficial to program understanding. In that case, the objective is compatible with that of program optimization (dead code elimination [2]), but this is certainly not the case in general. On the other hand, the replacement of (occurrences of) variables by their values is not so obvious.

The benefit depends on what these variables mean for the user: Variables like PI, TAX_RATE, etc., are likely to be kept in the code; on the contrary, intermediate variables used only to decompose some computations may be not so meaningful for the user, and he may prefer to have them removed.

Replacing variables by their values may lead to dead code (by making the assignments to these variables useless) and thus gives more opportunities to remove code. However, this is certainly not a sufficient reason to do systematic replacement. Of course, even when there is no replacement, the known value of a variable is kept in the environment of our simplification rules, as it can give opportunities to remove useless code, for instance, if the condition of an alternative may be evaluated thanks to that knowledge (and thus a branch may be removed).

The benefit of replacement depends not only on the kind of variable but also on the kind of user: A user who knows the application program well may prefer to keep the variables, the meaning of which is already known to him; a user trying to understand an unfamiliar application program may prefer to see as few variables as possible. In fact, our experiments have shown that the system must be very flexible in that respect. Thus, our system works as follows. There are three options: no replacement, systematic replacement, and each replacement depending on the user.

This paper is organized as follows. First, we justify our interest in scientific applications written in FORTRAN in Sec. 2. Next, we present in Sec. 3 the two main tasks of our partial evaluator: constant propagation and simplification. In Sec. 4, we specify our partial evaluation as a set of inference rules, and we show how these rules combine constant propagation and simplification rules. Section 5 presents proofs of soundness and completeness of our partial evaluation rules with respect to the dynamic semantics of FORTRAN. Section 6 explains how we implement our partial evaluator and gives some quantitative results. Section 7 presents conclusions and future work.

2. Scientific Programming

Many scientific application programs, written in FORTRAN for decades, are still vital in various domains (management of nuclear power plants, telecommunication satellites, etc.), even though more modern languages are used to implement their user interfaces. It is not unusual to spend several months to understand such application programs before being able to maintain them. For example, understanding an application program of 120,000 lines of FORTRAN code took nine months [12]. So, providing the maintainer with a tool, which finds parts of lost code semantics, allows him to reduce this period of adaptation.

2.1. *Characteristics*

One of the peculiarities of scientific applications is that the technological level of scientific knowledge (linear systems resolution, turbulence simulaton, etc.) is higher than the knowledge usually necessary for data processing (memory allocation, data

representations). The discrepancy is increased by the widespread use of FORTRAN, which is an old-fashioned language. Furthermore, for large scientific applications at EDF[a], FORTRAN 77 [1], which is quite an old version of the language, is used exclusively to guarantee the portability of the applications between different machines (mainframes, workstations, vector computers).

2.2. *General purpose applications*

Our study has highlighted common characteristics in FORTRAN programming at EDF. These scientific applications have been developed a decade ago. During their evolution, they had to be reusable in new various contexts. For example, the same thermohydraulic code implements both general design surveys for a nuclear power plant component (core, reactor, steam generator, etc.) and subsequent improvements in electricity production models. The result of this encapsulation of several models in a single large application domain increases program complexity, and thus amplifies the lack of structures in the FORTRAN programming language.

This generality is implemented by FORTRAN input variables whose value does not vary in the context of the given application. We distinguish two classes of such variables:

- *data about geometry*, which describes the modeled domain. They appear most frequently in assignment statements (equations that model the problem).
- *data taking a small number of values*, which are either *filters* necessary to switch the computation in terms of the context (modeled domain), or *tags* allowing us to minimize risks due to precision error in the output values. They appear in particular in conditions of alternatives or loops.

Figure 1 shows an example of program reduction. The code section in Fig. 1(a) is extracted from one of the application programs we have studied [20]. The partial evaluation of this code section according to the reduction criteria in Fig. 1(b) yields the code section in Fig. 1(c). In order to show how the reduced code has been obtained from the initial one, some links between both codes are shown. The initial code which is left unchanged in the simplified code is italicized. Expressions which have been replaced by their value and which appear in the reduced code are written in bold. The rest of the initial code is the code that has been removed in the reduced code. When using the tool, such links can be displayed in different colors.

A maintenance team is used to update a specific version of the application. These people know some filter properties (IC = 0 and IREX = 1) as well as data about geometry (DXLU = 0.5). Furthermore, IM is a tag whose value is 20. The knowledge of these values of input variables reduces the code (as shown in Fig. 1(c)). Because of the truth of the relation IREX = 1, two alternatives are simplified (1). The first alternative is simplified to its then-branch. In this branch, the variable

[a] EDF is the national French company that provides and distributes electricity to the whole country.

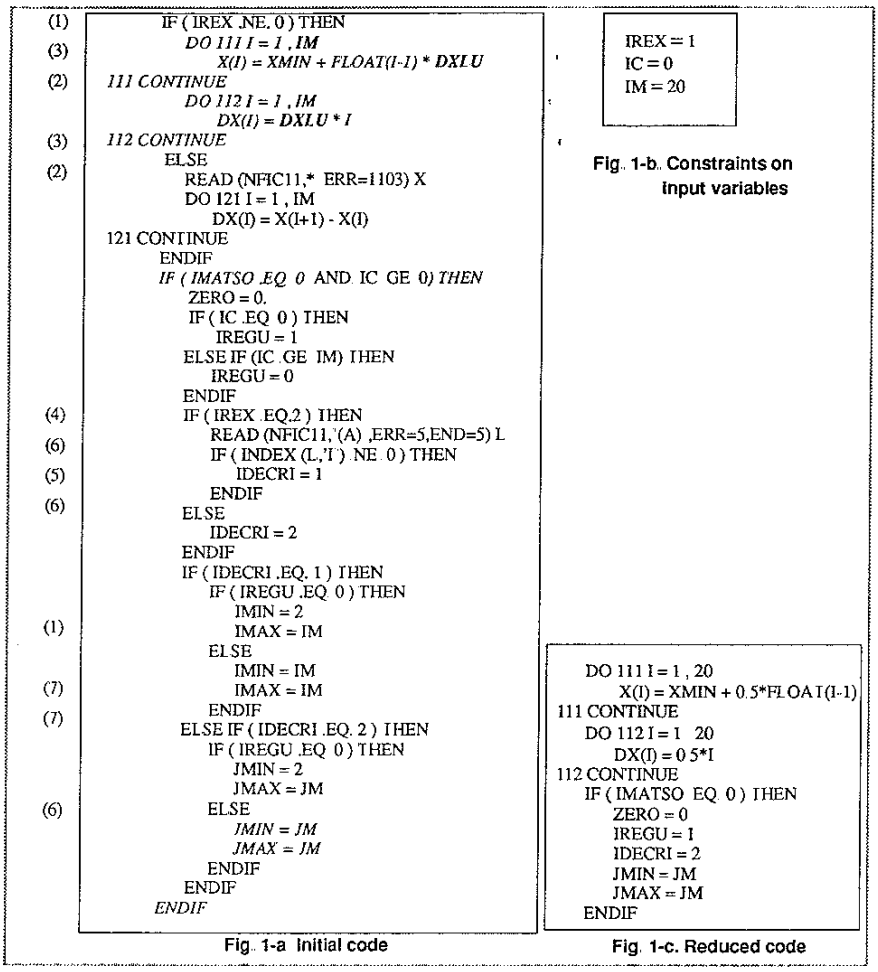


Fig. 1. An example of program reduction

DXLU is replaced by its value, which modifies the variables X(I) and DX(I) (2) The variable IM is also replaced by its value (3).

The condition of the next alternative is simplified (4) since the value of IC is known Furthermore, the relation IC = 0 simplifies the following alternative to its then-branch (5), which allows us to compute the value of the variables ZERO, IREGU, and IDECRI in the branch (6). Because the values of these three variables are constant values, the three corresponding assignments are removed from the code only if all possible uses of these occurrences of variables do not use another value of the variables (as discussed later) Then other alternatives are simplified (7).

3. Two Aspects of Partial Evaluation Applied to Imperative Programs

Our partial evaluator performs two main tasks: constant propagation through the code and simplification of statements. The tool can give the result of one of the two independently of the other. As we mentioned in the introduction, the simplification is up to the user. For instance, if he is a physicist who is familiar with the equations implemented in the code, he may wish to locate in FORTRAN statements these equations and their variables as they appear in the formulae of these equations (with the removal of unused statements but without the replacement of variables by their values). But if the user is a maintainer who does not know the application well, he would rather visualize the code simplified as much as possible. However, for optimal partial evaluation, the tool performs both tasks.

3.1. *Constant propagation*

Constant propagation is a well-known global flow analysis technique used by compilers. It aims at discovering values that are constant on all possible executions of a program and to propagate forward these constant values as far as possible through the program. Some algorithms now exist to perform fast and powerful constant propagation [25].

We describe in this section our constant propagation process, a version derived from [25]. It modifies most expressions by replacing some variable occurrences by their values and by normalizing all expressions through symbolic computation. Presently, our tool propagates only *equalities (and some specific inequalities) between variables and constants*. Of course, that limits the results of the analysis.

Substitution. Before running the partial evaluator, the user specifies numerical values for some input variables of the program (based on his personal knowledge of the application). Constant propagation spreads this initial knowledge supplied by the user. In the first stage, the partial evaluator replaces each specified variable by its value. Then, expressions whose operands are all constant values are evaluated and these resulting values are propagated forward through the whole program. This technique allows us to remove from the code all occurrences of variable identifiers that are no longer meaningful. The substituted values may be displayed differently from other values (in bold as in the previous example or in a different color).

Furthermore, the user can specify some variables for which the substitution will not be displayed. For instance, he can indicate that the variable PI will not be replaced by 3.1416 (of course, that value will nevertheless be used to possibly simplify statements).

Normalization. For any given numerical expression, we have to recognize if it reduces to a constant value (e.g., $x + 3 - x$ reduces to 3). In the same way, for any given logical expression, we have to recognize if it reduces to a conjunction of equalities or to a disjunction of inequalities. In the first case (respectively the second case), we will be able to propagate equalities in the then- (respectively else-) branch of alternatives. For example, if the condition of an alternative reduces to

($x = 1$ AND $y = 4$), it is propagated in the then-branch of the alternative. To do this, we perform constant propagation. To propagate constant values, our system normalizes each expression into a canonical form: a polynomial form for numerical expressions and a conjunctive normal form for logical expressions.

In a polynomial form, expressions are simplified by computing the values of the coefficients of the polynomial. Polynomial forms are written according to the decreasing powers order. When some terms of a polynomial have the same degree (e.g., z^2, x_3^2, and $t * u$), they are sorted according to the lexicographical order (e.g., $t * u < x^2 < z^2$). The canonical form of a relational expression is obtained from the canonical forms of its two numerical subexpressions. In normalized relational expressions, all variables and values occur on only one side of the operator (e.g., $x = y + 5$ is normalized to $x - y - 5 = 0$).

Because of these modifications of expressions, overflow, underflow, or round-off errors may happen. Therefore, the normalization of expressions may cause run-time errors. Conversely, some run-time errors may vanish thanks to the partial evaluation. As do most partial evaluation systems [18], our tool ignores such problems. In this regard alone, the tool does not yield a program which behaves like the initial one.

3.2. *Simplification*

Simplification is the second task of the partial evaluator. First, the expressions are simplified during the propagation as explained above. Then, the partial evaluator reduces the size of the code by removing statements which are never used for the specified values. This simplification includes the elimination of redundant tests and, in particular, the simplification of alternatives to one case thanks to the evaluation of their conditions. To simplify a statement means to remove it or to simplify its components. This section defines the simplification for each statement.

A *write* statement is simplified by simplifying those of its parameters that are expressions. A *read* statement is simplified by removing parameters whose values are known input values, and by adding an assignment for each parameter before the read statement. If all its parameters have known values, the read statement is removed.

An *assignment* simplification consists of simplifying the assigned expression.

An *alternative* is reduced to one of its branches when its condition has been evaluated to either true or false, and the branch is simplified. Otherwise, the statements of both branches are simplified. In this case a branch may become the empty statement.

Loops that are never entered are removed. When infinite loops are discovered, a comment is added at the beginning of the loop and its body is simplified. The only statements of a loop which are simplified by the knowledge of variable values are those statements whose expressions are invariant. Thus, we do not expand loops because we want to keep the original structure of the code. Since some FORTRAN

77 loops are implemented using labels and "go to" statements, when such a loop is removed, its label statement (at the entry of the loop) is kept while other statements contain "go to" statements to such labels (the label is left unchanged and the statement is replaced by an empty statement)

A *call* statement simplification consists of adding an assignment for each actual parameter of the call before the call statement. The identifier of the called subroutine is left unchanged in the current program (subroutines are not necessarily specialized) and the user has to run the partial evaluator on this subroutine code if it is also to be simplified. In this case, the tool provides him with the list of variables whose value is known at the entry point of the call.

The partial evaluation does not simplify other statements. This subset of FORTRAN being general enough, such a partial evaluation could be easily applied to other imperative languages. Note that our tool is as yet incapable of dealing with either inequalities (except for those appearing in conditions of alternatives) or literal values.

4. Inference Rules for Partial Evaluation

To specify the partial evaluation, we use inference rules operating on the FORTRAN abstract syntax and expressed in the natural semantics formalism [16], augmented with some VDM [14] operators. This section first presents rules defining both the constant propagation process and the simplification process. Then, it details the rules for partial evaluation of statements. These new rules combine the propagation rules and the simplification rules. Note that these techniques are not new, but we specify and use them in a novel way.

4.1. *Propagation and simplification rules*

In the following, we use sequents such as $H \overset{propag}{\vdash} I : H'$ (propagation), $H \overset{simpl}{\vdash} I \rightarrow I'$ (simplification), and a combination of both $H \overset{PE}{\vdash} I \longrightarrow I', H'$ (propagation and simplification). In these sequents:

- H is the environment associating values to variables whose values are known before executing I. It is modeled by a VDM-like map [14], shown as a collection of pairs contained in set braces such as {variable → constant, ...}, where no two pairs have the same first elements. Our system initializes such maps by the list of variables and their initial values, supplied by the user.
- I is a FORTRAN statement (expressed in a linear form of the FORTRAN abstract syntax)
- I' is the simplified statement under the hypothesis H.
- H' is H which has been modified by the execution of I.
- The superscript of the turnstile, such as *propag*, *simpl*, or *PE* denotes the set of rules the sequent belongs to.

In the sequents, we use the map operators *dom*, $\cup$, $\cap$, $=$, $\dagger$, $\lhd$, and $\lhd\!\!\!-$.

- The domain operator *dom* yields the set of first elements of the pairs in the map
- The union operator $\cup$ yields the union of maps whose domains are disjoint (in VDM, this operator is undefined if the domains overlap).
- The intersection operator $\cap$ of two maps yields the pair common to both maps.
- The equality operator $=$ of two maps yields true if and only if each pair of one map is a pair of the other map (and reciprocally)
- The map override operator $\dagger$ whose operands are two maps, yields a map which contains all of the pairs from the second map and those pairs of the first map whose first elements are not in the domain of the second map.
- The map restriction operator $\lhd$ is defined with a first operand which is a set and a second operand which is a map; the result is all of those pairs in the map whose first elements are in the set.
- When applied to a set and a map, the map deletion operator $\lhd$ yields those pairs in the map whose first elements are not in the set

The examples in Fig. 2 illustrate these definitions of map operators

$m = \{X \to 5, B \to true\}$	$dom(m) = \{X, B\}$
	$m \cup \{Y \to 7\} = \{Y \to 7, X \to 5, B \to true\}$
	$\{X, Z\} \lhd m = \{X \to 5\}$
	$\{B\} \lhd m = \{X \to 5\}$
	$m \cap \{X \to 5, B \to false\} = \{X \to 5\}$
$n = \{C \to false, X \to 8\}$	$m \dagger n = \{X \to 8, B \to true, C \to false\}$
	$n \dagger m = \{X \to 5, B \to true, C \to false\}$

Fig. 2. Some map operators

We have written some rules to explain how sequents are obtained from other sequents. A rule has two parts, separated by a horizontal bar. Above the bar is a set of sequents, which are the premises of the rule. This set may be empty (in which case no bar is drawn). Below the bar is a single sequent, the conclusion of the rule. If the premises hold, then the conclusion holds. The subscript on a turnstile is omitted when the type of the sequent is evident from the context.

The rules we present in Fig. 3 express the simplification of logical or numerical expressions. They belong to the *eval* system, which is a subsystem of the simplification system *simpl*. The first rule has no premise. It specifies that a variable X which belongs to the environment is simplified into a constant which is equal to its value C. Otherwise (second rule), the variable is not modified

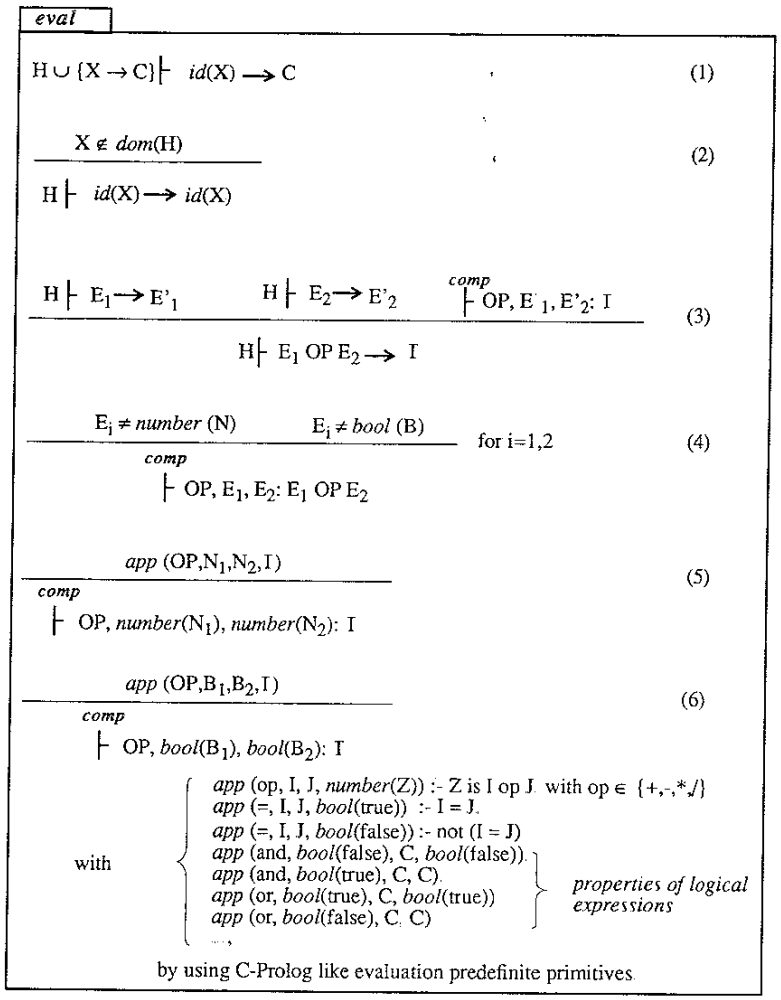


Fig. 3. Simplification of expressions

To evaluate an expression E_1 OP E_2 to the value T, its two operands E_1 and E_2 must have been evaluated to E'_1 and E'_2 respectively, and the value T is the result of the computation of E'_1 OP E'_2 (through the *comp* system that performs numerical computation) (third rule). If E'_1 and E'_2 are both constants (respectively N_1 and N_2), the computation of T is processed by the application of the *app* primitive to the operator OP and to its two operands N_1 and N_2 (fifth and sixth rules). Otherwise, the result of the computation is E'_1 OP E'_2 (fourth rule). Some classical properties of logical expressions have been enclosed in the *app* primitive to perform a deeper evaluation of expressions. The *app* primitive presumes correct typing of programs.

The rule in Fig 4(a) expresses the propagation through a sequence of statements. To propagate the environment H through the sequence of statements I_1; I_2, H is propagated through the statement I_1, which updates H to H_1. This new environment H_1 is propagated through I_2, which updates H_1 to H_2.

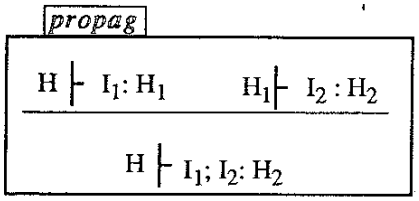


Fig. 4(a). Propagation through a sequence of statements

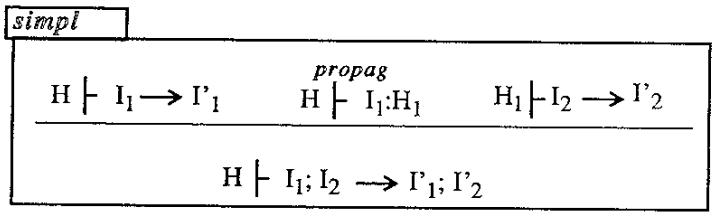


Fig. 4(b). Simplification of a sequence of statements

The rule in Fig. 4(b) expresses the simplification of such a sequence. Given an environment H, to simplify a sequence of statements I_1; I_2, the first statement I_1 is simplified to I'_1, and the environment H is propagated through I_1. In this new environment H_1, the second statement I_2 is then simplified to I'_2. I'_1; I'_2 is the simplified sequence of statements.

Figure 5 presents six of the eight simplification and propagation rules for alternatives. If the condition C of an alternative evaluates to true, then:

- the environment H' resulting from the propagation of H through the alternative is obtained by propagating H through the statements I_1 of the then-branch (first rule: propagation),
- the simplification of the alternative is the simplification of its then-branch (second rule: simplification).

In the same way, there are two rules for an alternative whose condition evaluates to false (in these rules "true" becomes "false", "I_1" becomes "I_2", and "then" becomes "else"). Since these rules are very similar to the first two rules, they do not appear in Fig. 5. They are shown with partial evaluation rules in Fig. 8.

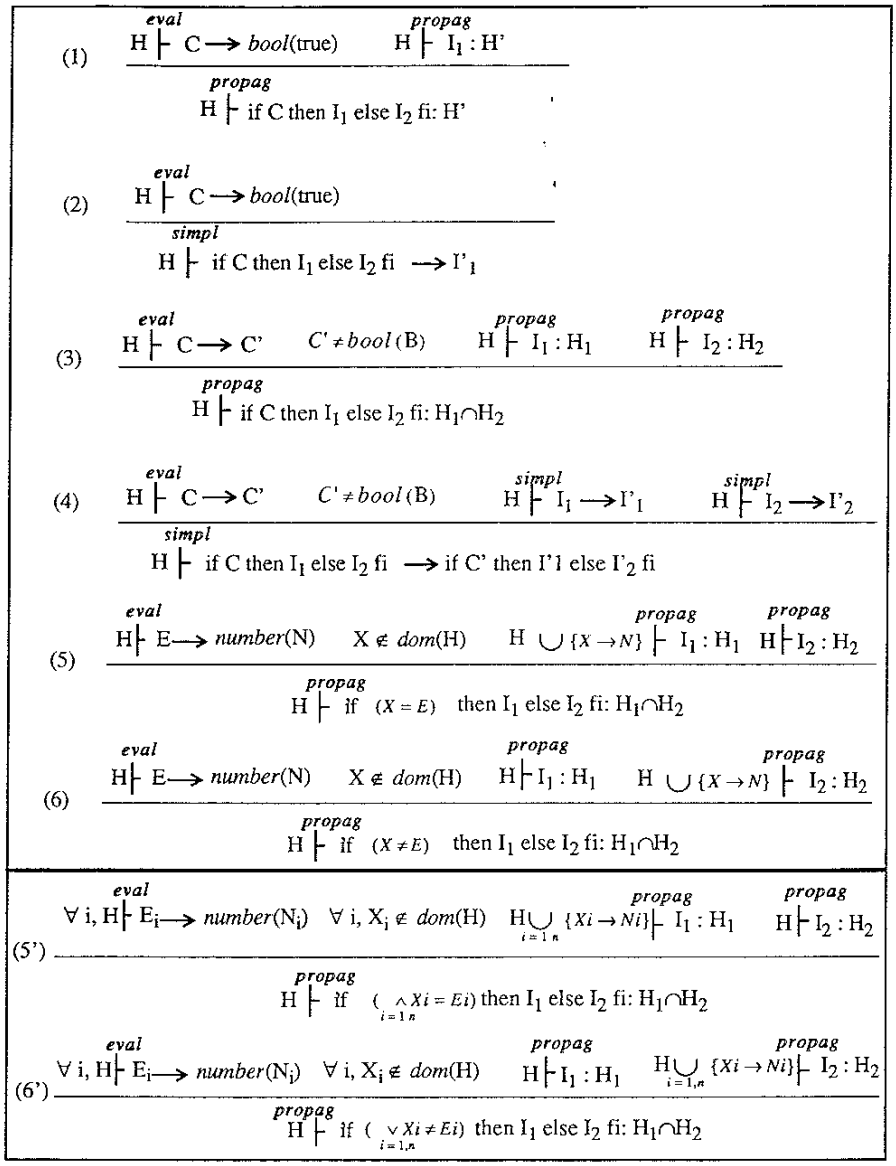


Fig. 5. Propagation and simplification rules for alternatives

If the condition C of an alternative is only partially evaluated to C', the propagation and simplification proceed along both branches of the alternative:

- the propagation of H through the then-branch I_1 (respectively the else-branch I_2) leads to an environment H_1 (respectively H_2). The intersection of both en-

vironments is the final environment: If a variable has the same value in both environments H_1 and H_2, that value is kept in the final environment, otherwise it is removed from the final environment (third rule: propagation).

- the simplification of the alternative yields the alternative whose condition is the partially evaluated condition C' and whose branches are the simplified branches of the initial alternative (fourth rule: simplification)

The fifth rule in Fig. 5 is a propagation rule. It shows that information can sometimes be derived from the equality tests that control alternatives. If the condition of an alternative is expressed as an equality such as $X = E$, where X is a variable that does not belong to the domain of the environment H and E evaluates to a constant N, then the pair (X, N) is added to the environment related to the then-branch.

Since the statements "*if* $X \neq E$ *then* I_1 *else* I_2 *fi*" and "*if* $X = E$ *then* I_2 *else* I_1 *fi*" are semantically equivalent, there is a corresponding rule (sixth rule) for a condition of an alternative expressed as an inequality such as $X \neq E$. In that case, the pair (X, N) is added to the environment related to the else-branch. Rules 5 and 6 express that only equalities between variables and constants can be added to the environment. Thus, if other information is expressed in the condition, it is not taken into account by the partial evaluator.

Rules 5 and 6 have been generalized to conditions of alternatives expressed as conjunctions of equalities and disjunctions of inequalities (rules 5' and 6') In these rules, we have used generalized AND (noted $\underset{i=1,n}{\wedge}$ and OR (noted $\underset{i=1,n}{\vee}$).

By adding the unfolding of loops, the dynamic semantics would be a special case for our system: If the initial environment associates values to all input variables, the final environment would give (among others) the values of all output variables, with the executable part of the program simplified to the empty statement. Thus, the very special use of the implementation of our system is to see it as a standard interpreter.

Since the simplification is performed in the context of the propagation, and the propagation uses the simplification of expressions, we have chosen to combine propagation and simplification in our rules.

4.2. *Combined Rules*

For every FORTRAN statement, we have written rules that describe the combination of the propagation and simplification systems The combination $\overset{PE}{\longrightarrow}$ of these two systems is defined by:

$$H \vdash I \overset{PE}{\longrightarrow} I', H' \quad \text{iff} \quad H \overset{propag}{\vdash} I : H' \quad \text{and} \quad H \overset{simpl}{\vdash} I \rightarrow I'$$

From this rule, we may define inductively the $\overset{PE}{\longrightarrow}$ relation. For instance, Fig. 6 shows the rule for a sequence of statements. A sequence is evaluated from left to right: The partial evaluation of a sequence of two statements I_1 and I_2 consists

of simplifying I_1 to I'_1 and then updating (adding, deleting, or modifying) the environment H. In this new environment H_1, I_2 is simplified to I'_2 and H_1 is modified to H_2.

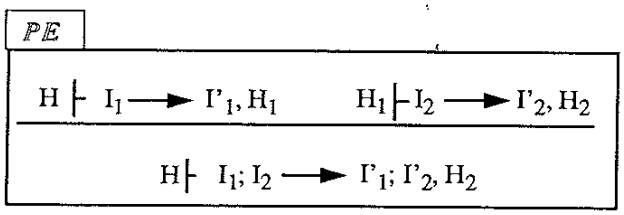


Fig. 6. Partial evaluation of a sequence of statements.

Figure 7 specifies the rules for partial evaluation of assignments. The *eval* notation refers to the formal system of rules which simplifies the expressions that we have previously presented in Fig. 3.

If the expression E evaluates to a numerical constant N, the environment H is modified: the value of X is N whether X already has a value in H or not. With the kind of propagation performed, the assignment X := E can be removed only if all possible uses of that occurrence of X do not use another value of X. For instance, in the sequence

$$X := 2;\ \text{if CODE} \neq 5 \text{ then } X := X + 1 \text{ fi};\ Y := X,$$

the value 2 of X is propagated in the expression X + 1, but the assignment X := 2 cannot be removed because in the assignment Y := X, X comes from either from X := 2 (value 2) or from X := X + 1 (value 3). Thus, that sequence is only simplified into X := 2; if CODE ≠ 5 then X := 3 fi; Y := X.

To eliminate assignments that become useless after the partial evaluation, we use classical dead code elimination algorithms [2]. Thus, elimination of redundant assignments is performed in a separate optimization phase.

If E is only partially evaluated to E', the expression E is modified as part of the assignment X := E and the variable E is removed from the environment if it was in it, because its value has become unknown.

The following examples illustrate these two cases. In *Ex. 1*, as the value of the variable A is known, the new value of the assigned variable C is introduced into the environment. We suppose that the assignment C := A + 1 can be removed from the reduced program. In *Ex. 2*, after the partial evaluation of the expression A + B, the value of C becomes unknown. Such a case only happens when A and B do not have both constant values.

Ex. 1 $\{A \to 1, C \to 4\} \vdash C := A + 1 \longrightarrow \text{skip}, \{A \to 1, C \to 2\}$

Ex. 2 $\{A \to 1, C \to 2\} \vdash C := A + B \longrightarrow C := 1 + B, \{A \to 1\}$

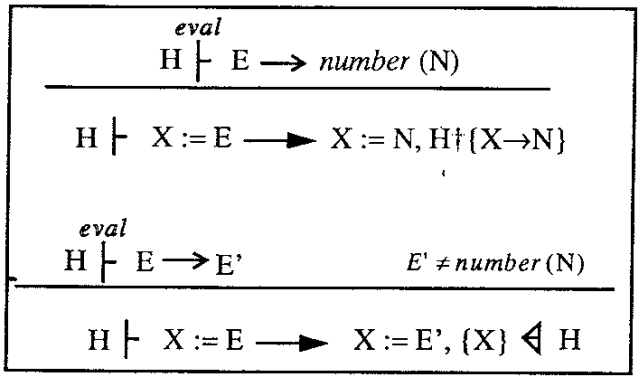


Fig. 7. Partial evaluation of assignments

The rules for partial evaluation of alternatives are defined in Fig. 8. If the condition C of an alternative evaluates to a logical constant, this alternative can be simplified to the corresponding simplified branch. If C is only partially evaluated to C', the partial evaluation proceeds along both branches of the alternative, and the final environment is the intersection of the two environments resulting from the simplification of both branches (as explained above).

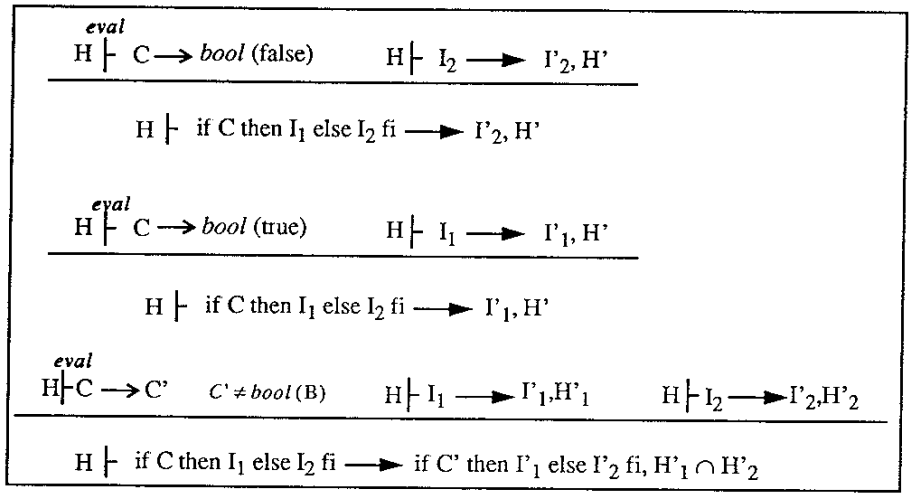


Fig. 8. Partial evaluation of alternatives

As for alternatives, the rules for partial evaluation of loops, presented in Fig. 9, depend on the ability to evaluate the truth or falsity of the condition C in the current environment H. The first rule specifies that if the loop is not entered, it is removed from the code. There is no specific rule for the case where C evaluates to true, because we do not expand loops.

In the second rule, if C evaluates to C' (and C' differs from false), the statements I of the loop can be simplified, given H is restricted to a loop invariant Inv(I). Inv(I) is a pessimistic estimation of the variables that are not modified by the loop body. It is calculated by the partial evaluator and consists of a list of variables whose values are neither on the left-hand side of an assignment, nor a modified parameter of a call or a parameter of a read statement. Thus, performing the propagation through I will not modify that restricted environment.

Some infinite loops are treated by the second rule: a loop is infinite when $\mathrm{Inv(I)} \lhd \mathrm{H} \overset{eval}{\vdash} \mathrm{C} \rightarrow \mathrm{bool(true)}$; in this case a comment is added at the beginning of the loop. Figure 9 shows the rules for while-statements, but similar rules exist for repeat-statements.

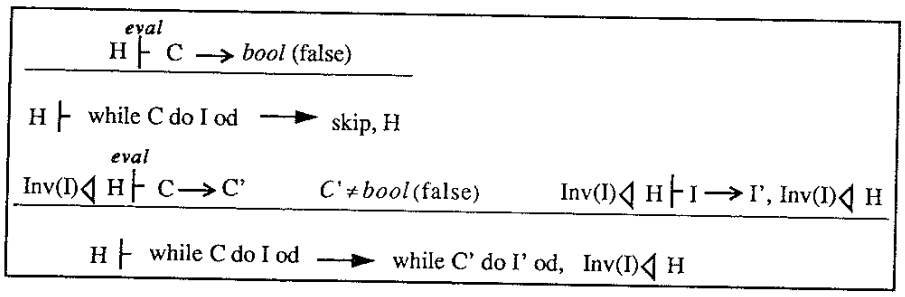


Fig. 9. Partial evaluation of while-loops

Some empty statements may be located in an initial program, for example, in an alternative with no then-branch. Figure 10 expresses that the partial evolution of an empty statement never affects the current environment H.

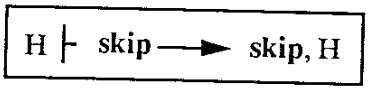


Fig. 10. Partial evaluation of an empty statement

Some partial evaluation rules have not been presented in this section. These are either rules allowing us to reach statements from an initial program (containing

other syntactical objects we do not analyze, such as data declarations) or rules performing the normalization of expressions using mainly associativity, distributivity, and reduction to the common denominator.

5. Soundness and Completeness of the Partial Evaluation

Our aim in this section is to show how to prove that the simplification presented above is correct with respect to the dynamic semantics of FORTRAN, given in the natural semantics formalism. We recall that partial evaluation of a program P with respect to input variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ for the values $x_1 = c_1, \ldots, x_m = c_m$ must give a residual program P', whose input variables are $y_1, \ldots, y_n$ and the executions of $P(c_1, \ldots, c_m, y_1, \ldots, y_n)$ and $P'(y_1, \ldots, y_n)$ produce exactly the same results.

We will show that this is expressed by two inference rules, one expressing soundness (each result of P' is correct with respect to P) and one expressing completeness (each correct result is computed by P' too). As P and P' are deterministic, we could have only one rule using equality, but the demonstration of our two rules is less complicated and more general (being also applicable for nondeterministic programs). Examples of proofs for some FORTRAN statements are detailed in this section.

5.1. *Rules proving soundness and completeness*

To prove the simplification, we need a formal dynamic semantics of FORTRAN and we must prove the soundness and completeness of the simplification rules with respect to that dynamic semantics.

To express the dynamic semantics of FORTRAN, we use the same formalism (natural semantics [16]) as for simplification. Natural semantics has its origin in the work of Plotkin [13, 21]. Under the name "structured operational semantics", he gives inference rules as a direct formalization of an intuitive operational semantics: His rules define inductively the transitions of an abstract interpreter. Natural semantics extends that work by applying the same idea (use of a formal system) to different kinds of semantic analysis (not only interpretation, but also typing, translation, etc.). Thus, the semantic rules we give have to generate theorems of the form

$$\mathrm{H} \overset{sem}{\vdash} \mathrm{I} : \mathrm{H'},$$

meaning that, in environment H, the execution of statement I leads to the environment H' (or the evaluation of expression I gives value H'). These rules are themselves not proved: they are supposed to define ex nihilo the semantics of FORTRAN, as Plotkin [21] and Kahn [16] did for other languages like ML.

To prove these rules would mean having another formal semantics (e.g., a denotational one) and to prove that the rules are sound and complete with respect to it. But there is no official semantics for FORTRAN. Thus, that proof would have to be proof of consistency between two dynamic semantics we give. That is outside

the scope of our work. We want to prove consistency between simplification and dynamic semantics, not between two dynamic semantics.

Now how can we prove that the simplification system is sound and complete with respect to the dynamic semantics system? Instead of the usual situation, that is a formal system and an intended model, we have two formal systems: the simplification system (noted *PE*) and the dynamic semantics system (noted *sem*) A program P is simplified to P' under hypothesis H_0 on some input variables if and only if

$$H_0 \vdash P \rightarrow P'$$

is a theorem of the simplification system

Let us call H the environment containing the values of the remaining input variables Thus, $H_0 \cup H$ is the environment containing the values of all input variables With that initial environment, P' (respectively P) evaluates to H' if and only if

$$H_0 \cup H \overset{sem}{\vdash} P'\text{: } H' \text{ (respectively } H_0 \cup H \overset{sem}{\vdash} P\text{:}H')$$

is a theorem of the dynamic semantics (*sem*) system.

Now, soundness of simplification with respect to dynamic semantics means that each result computed by the residual program is computed by the initial program That is, for each P, P', H_0, H, H', if P is simplified to P' under hypothesis H_0 and P' executes to H' under hypothesis $H_0 \cup H$, then P executes to H' under hypothesis $H_0 \cup H$. Thus, soundness of simplification with respect to dynamic semantics is formally expressed by the first rule in Fig 11.

Completeness of simplification with respect to dynamic semantics means that each result computed by the initial program P is computed by the residual program P'. Thus, it is expressed by the second inference rule in Fig. 11. In fact, our approach to prove simplification is very close to the approach of [8] to prove the correctness of translators. In that paper, dynamic semantics and translation are both given by formal systems, and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules (that are proved by induction on the length of the proof; here we will use rule induction instead)

Note that both rules are not the most restricting rules (for instance, their initial environment is $H_0 \cup H$ and not only H, to allow partial simplification).

To prove both rules concerning programs in Fig. 11, we prove that they hold for any statement we partially evaluate (remember that we do not analyze data declarations). Thus, we have to prove that both rules in Fig. 12 hold. In these rules, I denotes a FORTRAN statement and I' denotes the corresponding simplified statement

The dynamic semantics of FORTRAN formalized by the *sem* system is shown in Fig. 13. Some dynamic semantics rules (such as the rules for alternatives) are propagation rules For that reason, in the *sem* system, we have overloaded the

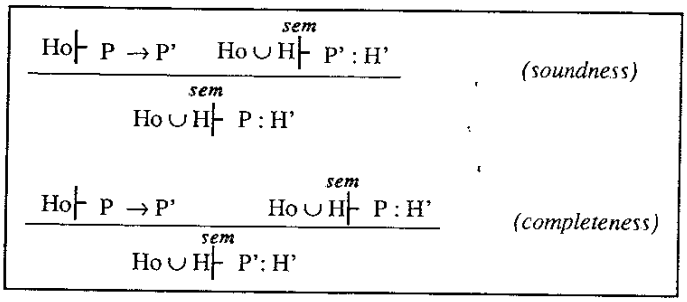


Fig. 11. Soundness and completeness of the program simplification

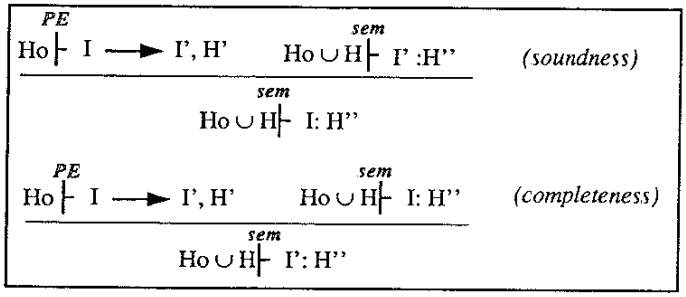


Fig. 12. Soundness and completeness of the statements partial evaluation

":" symbol, representing the system *propag*, instead of using a new symbol. The dynamic semantics rules for loop statements differ from the corresponding propagation rules since we do not expand loops while performing the partial evaluation. The interpretation of loop statements is made here by unfolding.

To prove the validity of the completeness and soundness rules, we use rule induction on the partial evaluation and on the dynamic semantics. Indeed, the *PE* and *sem* systems have been defined inductively.

Our inductive hypothesis for soundness is the following property Π_s, which is defined as follows: $\Pi_s\ (H_0, I, I', H') \Leftrightarrow$

$$(\forall\, H_0,\, I, I', H' \mid H_0 \overset{PE}{\vdash} I \longrightarrow I', H' : (\forall\, H, H''\ H_0 \cup H \overset{sem}{\vdash} I' : H'' \Rightarrow H_0 \cup H \overset{sem}{\vdash} I : H''))$$

The inductive hypothesis Π_c for completeness is defined in a similar way. The rule induction on *PE* (respectively *sem*) states that quadruples are only obtained by the rules belonging to the *PE* (respectively *sem*) system.

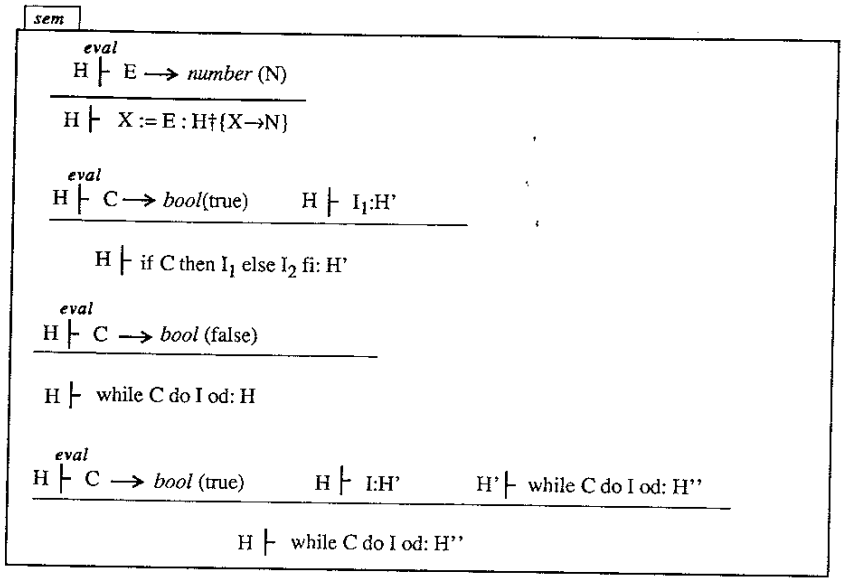
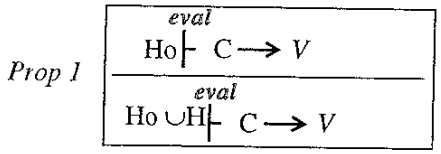


Fig 13. Some FORTRAN dynamic semantics rules.

To construct our proof trees we use property *Prop 1*, which states that if some (variable, value) pairs are added to an environment H_0, what had already been proved in this environment H_0 still holds in the new environment $H_0 \cup H$.



5.2. *Examples of proofs of soundness*

The following examples deal only with proofs of soundness. Proofs of completeness are similar. We start with treating simple statements, which are not composed of other statements. They form the basic cases of proof. Figure 14 shows a proof of such a statement. The possible removal of assignment does not appear in the proof tree, since it is performed during a dead code elimination phase, subsequent to the evaluation of the expression of the assignment.

Once simple statements have been proved, we have to prove that the soundness rule holds for composite statements. Figure 15 shows a proof of soundness for an alternative whose condition evaluates to true. There is a similar proof for the case when the condition evaluates to false.

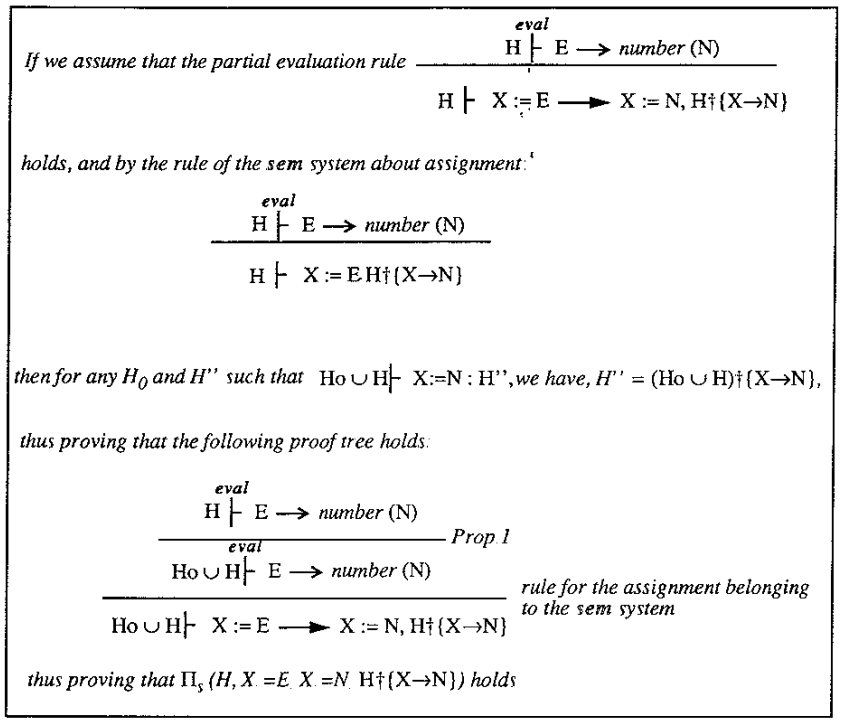


Fig. 14. Proof of soundness of an assignment partial evaluation

6. Implementation

This section describes the overall architecture of our system. Then, it gives quantitative results.

6.1. *Architecture of the partial evaluator*

The partial evaluation rules are very close to the ones we have implemented in the Centaur/FORTRAN environment. The Centaur system [6] is a generic programming environment parameterized by the syntax and semantics of programming languages. When provided with the description of a particular programming language, including its syntax and semantics, Centaur produces a language specific environment. The resulting environment consists of a structured editor, an interpreter/debugger, and other tools, together with a uniform graphical interface. Furthermore, in Centaur, program texts are represented by abstract syntax trees. The textual (or graphical) representation of abstract syntax tree nodes may be specified by pretty-printing rules. Centaur provides a default representation.

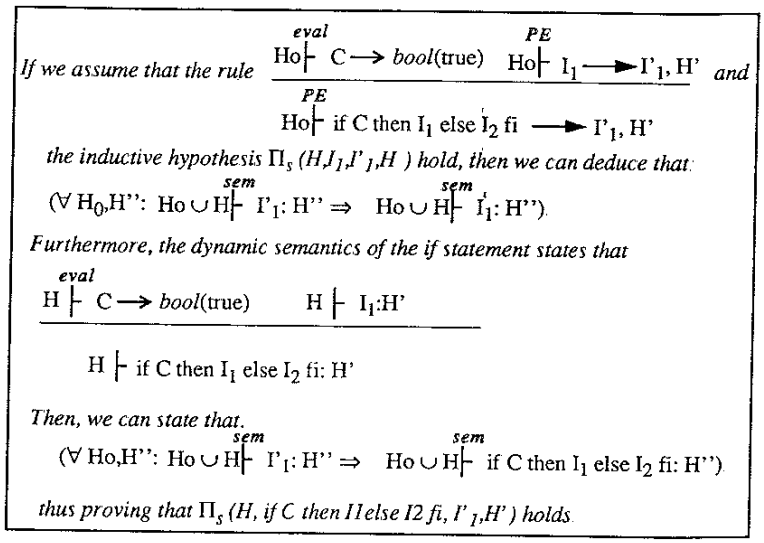


Fig. 15. Proof of soundness of an alternative whose condition evaluates to true

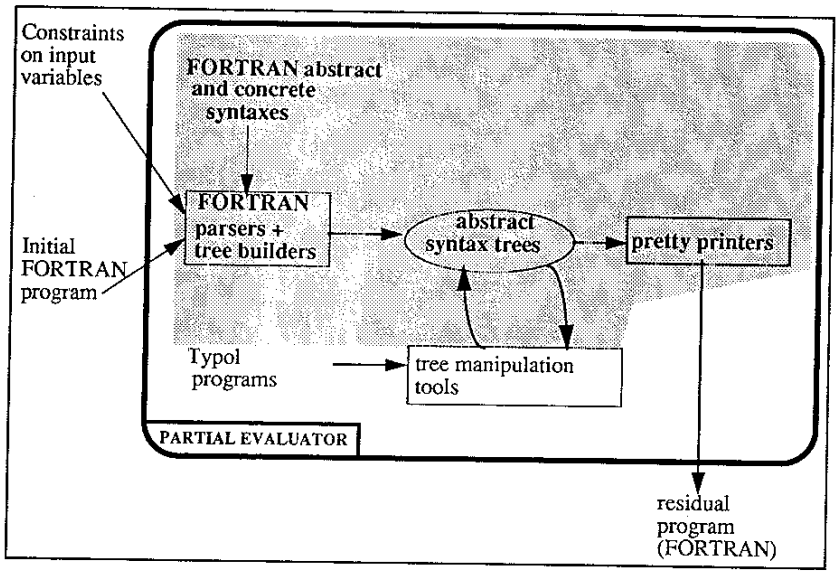


Fig. 16. The Centaur/FORTRAN environment

We have used such a resulting environment, Centaur/FORTRAN, to build our partial evaluator. From Centaur/FORTRAN, we have implemented an environment for partial evaluation of FORTRAN programs. Figure 16 shows the overall architecture of this environment, where Centaur/FORTRAN is represented by the grey part.

A language for specifying the semantic aspects of languages called Typol is included in Centaur, so that the system is not restricted to manipulations that are based solely on syntax. Typol is an implementation of natural semantics. It can be used to specify and implement static semantics, dynamic semantics, and translations. Typol specifications are compiled into Prolog code. When executing these specifications, Prolog is used as the engine of the deductive system.

6.2. *Quantitative results*

We have written about 200 Typol rules to implement our partial evaluator. Ten rules express how to reach abstract syntax nodes representing simplifiable statements. Ninety rules perform the normalization of expressions. Among the 100 rules for simplification, 60 rules implement the simplification of expressions. The 40 other rules implement the statements simplification. We have written about 25 Prolog predicates to implement the VDM operators we have used to specify the simplification. Thus, these operators are used in Typol rules as in the formal specification of the simplification.

The partial evaluator may analyze any FORTRAN program, but it simplifies only a subset of FORTRAN 77. This subset is a recommended standard at EDF. For instance, it does not analyze any "go to" statement (they are not recommended at EDF), but only "go to" statements that implement specific control structures (e.g., a while-loop).

The average initial length of programs or subroutines we have analyzed is 100 lines of FORTRAN code, which is lengthier than the recommended length at EDF (60–70 lines). The reduction rate amounts from 25% to 80% of lines of code. This reduction is especially important when there is a large number of assignments and conditionals. This is the case for most subroutines implementing mathematical algorithms. For subroutines whose main purpose is editing results or calling other subroutines, the reduction is generally not so important.

7. Conclusion

We have used partial evaluation for programs which are difficult to maintain because they are too general. Specialized programs for some values of their input variables are obtained by propagating these constant values (through a normalization of the expressions) and by performing simplifications on the code, for instance, alternatives are reduced to one of their branches. We have shown how to prove by rule induction the completeness and soundness of our formal system for partial evaluation, given the dynamic semantics of FORTRAN.

This technique of partial evaluation helps the maintainer to understand the program behavior in a particular context. Our experiments have given satisfactory results. The residual program is more efficient because many statements and variables have been removed, and no additional statement has been inserted. Another advantage of this technique is that it can also be applied to abstractions at a higher level than the code (e.g., it can be applied to algorithms). Note that the techniques we develop are not new, but we specify (inference rules), implement (Centaur), and use them (for program understanding) in a novel way.

Our partial evaluation system may be used in two ways: by visual display of the residual program as part of the initial program (for documentation or for debugging), or by generating this residual program as an independent (compilable) program. Our system has been tested on several examples and has given satisfactory results.

We are now focusing on the possibility for the user to supply general properties about input variables as in parameterized partial evaluation [10]. These general properties are, for instance, relational expressions composed of some literal values (e.g., $x < z + 4$) instead of only equalities to constant values. We will consequently take into account that kind of information in the conditions of alternatives and loops. We intend to apply linear resolution methods and symbolic manipulation packages for FORTRAN [7] to propagate such properties.

References

1. *FORTRAN*, ANSI standard X3.9 (1978).
2. A. Aho, R. Sethi, and J. Ullman, *Compilers* (Addison-Wesley, 1986).
3. V. Ambriola, F. Giannotti, D. Pederschi, and F. Turini, "Symbolic semantics and program reduction", *IEEE TOSE* **11**, 8 (1985) 784–794.
4. L. O. Andersen, "C program specialization", Master's Thesis, University of Copenhagen, Denmark (May 1992).
5. A. Berlin and D. Weise, "Compiling scientific code using partial evaluation", *Computer* **23**, 12 (1990) 25–37.
6. *Centaur 1.1 Documentation*, INRIA (January 1990).
7. P. D. Coward, "Symbolic execution systems — a review", *Softw. Eng. J.* (November 1988) 229–239.
8. J. Despeyroux, "Proof of translation in natural semantics", *Symp. Logic in Computer Science*, Cambridge, USA (June 1986).
9. A. P. Ershov and B. N. Ostrovski, "Controlled mixed computation and its application to systematic development of language-oriented parsers", *Program Specification and Transformation, IFIP'87* (1987) pp. 31–48.
10. C. Consel and S. C. Khoo, "Parameterized partial evaluation", *ACM TOPLAS* **15**, 3 (1993) 463–493.
11. K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance", *IEEE TOSE* **17**, 8 (1991) 751–761.
12. M. Haziza, J. F. Voidrot, E. Minor, L. Pofelski, and S. Blazy, "Software maintenance: an analysis of industrial needs and constraints", *IEEE Conf. Software Maintenance*, Orlando, USA (November 1992) pp. 18–26.
13. M. Hennessy, *The Semantics of Programming Languages* (Wiley, 1990).

14. C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed. (Prentice-Hall, 1990).
15. N. D. Jones, P. Sestoft, and H. Sondergaard, "MIX: a self-applicable partial evaluator for experiments in compiler generation", *Lisp and Symbolic Computation* **2** (1989) 9–50.
16. G. Kahn, "Natural semantics", *Proc. STACS'87, Lecture Notes in Computer Science*, Vol. 247 (March 1987).
17. V. Kasyanov, "Transformational approach to program concretization", *Theoretical Computer Science* **90** (1991) 37–46.
18. R. Kemmerer and S. Eckmann, "UNISEX: a UNIX-based symbolic executor for Pascal", *Software Practice and Experience* **15**, 5 (1985) 439–457.
19. U. Meyer, "Techniques for partial evaluation of imperative languages", in *Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, *ACM SIGPLAN Notices* **26**, 9 (1991) pp. 94–105.
20. G. Nicolas *et al.*, "A finite volume approach for 3D two phase flows in tube bundles: the THYC code", Kernforschungszentrum, Karlsruhe, Vol. 2 (1989) pp. 1247–1253.
21. G. Plotkin, "A structural approach to operational semantics", Technical Report DAIMI FN-19, University of Aarhus, Denmark (1981).
22. D. Sahlin, "An automatic partial evaluator for full Prolog", Ph.D. Thesis, SICS, Copenhagen, Denmark (March 1991).
23. T. H. Sneed, "The myth of 'top-down' software development and its consequences", *IEEE Conf. Software Maintenance*, Miami, USA (October 1989) pp. 22–29.
24. H. J. Van Zuylen, "Understanding in reverse engineering", *The REDO Handbook* (Wiley, 1992).
25. M. N. Wegman and K. Zadeck, "Constant propagation with conditional branches", *ACM TOPLAS* **13**, 2 (1991) 181–210.

Automated Software Engineering, 7, 345–376, 2000

Specifying and Automatically Generating a Specialization Tool for Fortran 90

SANDRINE BLAZY blazy@iie.cnam.fr
CEDRIC IIE, 18 allee Jean Rostand, 91 025 Evry Cedex, France

Abstract. Partial evaluation is an optimization technique traditionally used in compilation. We have adapted this technique to the understanding of scientific application programs during their maintenance. We have implemented a tool that analyzes Fortran 90 application programs and performs an interprocedural pointer analysis. This paper presents a dynamic semantics of Fortran 90 and manually derives a partial evaluator from this semantics. The tool implementing the specifications is also detailed. The partial evaluator has been implemented in a generic programming environment and a graphical interface has been developed to visualize the information computed during the partial evaluation (values of variables, already analyzed procedures, scope of variables, removed statements, etc.).

Keywords: program understanding, partial evaluation, dynamic semantics, formal specification, interprocedural analysis, alias analysis, proof of correctness

1. Introduction

A wide range of software maintenance tools analyze existing application programs in order to transform them (Baxter et al., 1998; Sellink and Verhoef, 2000; van den Brand et al., 1996; Yank et al., 1997). Some of these transformations aim at facilitating the understanding of programs. Furthermore, these transformations are based on rather complex analyses. As software maintenance tools, these tools must introduce absolutely no unforeseen changes in programs.

To ensure that the transformation preserves the meaning of programs, we have used formal specifications to develop a software maintenance tool. In our framework, a formal specification yields:

- A basis for expressing precisely which transformations are performed. Formal concepts are powerful enough to clarify concepts of programming languages and to model complex transformations. The formal specification can be seen as a reference document between specifiers and end-users. In our context, end-users are software maintainers who have a strong background in mathematics. Thus, they are disposed to understand our formal specifications.
- A mathematical formalism for proving and validating properties of program transformations.
- A framework for simplifying the implementation of a tool. The formal specification has been refined to obtain an implementation.

Our tool aims at improving the understanding of scientific application programs. Such programs are difficult to maintain mainly because they were developed a few decades ago by experts in physics and mathematics, and they have become very complex due to extensive modifications. For a maintenance team working on a specific application program, one of the most time consuming steps is to extract by hand in the code the statements corresponding to the specific context of the maintenance team. This context is very well known by all the people belonging to the maintenance team; this is their minimum knowledge concerning the data of their application program. This context is described by equalities between specific variables and values (Blazy and Facon, 1998).

1.1. Motivations

Partial evaluation is an optimization technique, also known as program specialization. When given a program and known values of some input data, a partial evaluator produces a so-called residual or specialized program. Running the residual program on the remaining input data will yield the same result as running the original program on all of its input data (Jones et al., 1993). The residual program is more optimized than the original program. Partial evaluation has been applied to generate compilers from interpreters (by partially evaluating the interpreter for a given program). In this context, previous work (ACM, 1998; Danvy et al., 1996) has primarily dealt with functional and declarative languages.

Partial evaluation has also been applied to improve speedups of functional, declarative and imperative programs (Andersen, 1994; Baier et al., 1994; Marlet et al., 1997). Usually, the chief motivation for doing partial evaluation is speed. The residual program is faster than the initial one because expensive calculations have been eliminated. This is done with sophisticated techniques such as binding-time analysis. The basic partial evaluation techniques are procedure inlining (calls to procedures are replaced with copies of their bodies) and loop unfolding (loops are replaced with several copies of their bodies). Both techniques replace statements by faster statements, but they increase the size of the code and generate new variables or rename variables. These partial evaluation techniques conflict with our goal, so we do not use them.

Our goal is to generate programs that are easier to understand. Thus, our transformations do not modifiy the structure of the code. In like manner, our partial evaluator neither generates new variables nor renames variables. Our partial evaluation is based on constant propagation and statements simplification. The programs we generate are easier to understand because many statements and variables have been removed and no additional statement or variable has been inserted. Furthermore, the known values of variables like PI or TAX RATE are propagated during partial evaluation but these variables are likely to be kept in the code (e.g. in the code 2*PI + 1 should be easier to understand than 7.28). The benefit of replacing variables by values depends also on the kind of user (see (Blazy and Facon, 1994) for details about our specialization strategy).

Figure 1 briefly illustrates how source code is specialized with respect to constraints on input variables. The source code that has been removed is striked out (e.g. in figure 1, the first `if` statement is removed and replaced by its specialized `then` branch). The source code that is not striked out corresponds to the specialized code. In the specialized code,

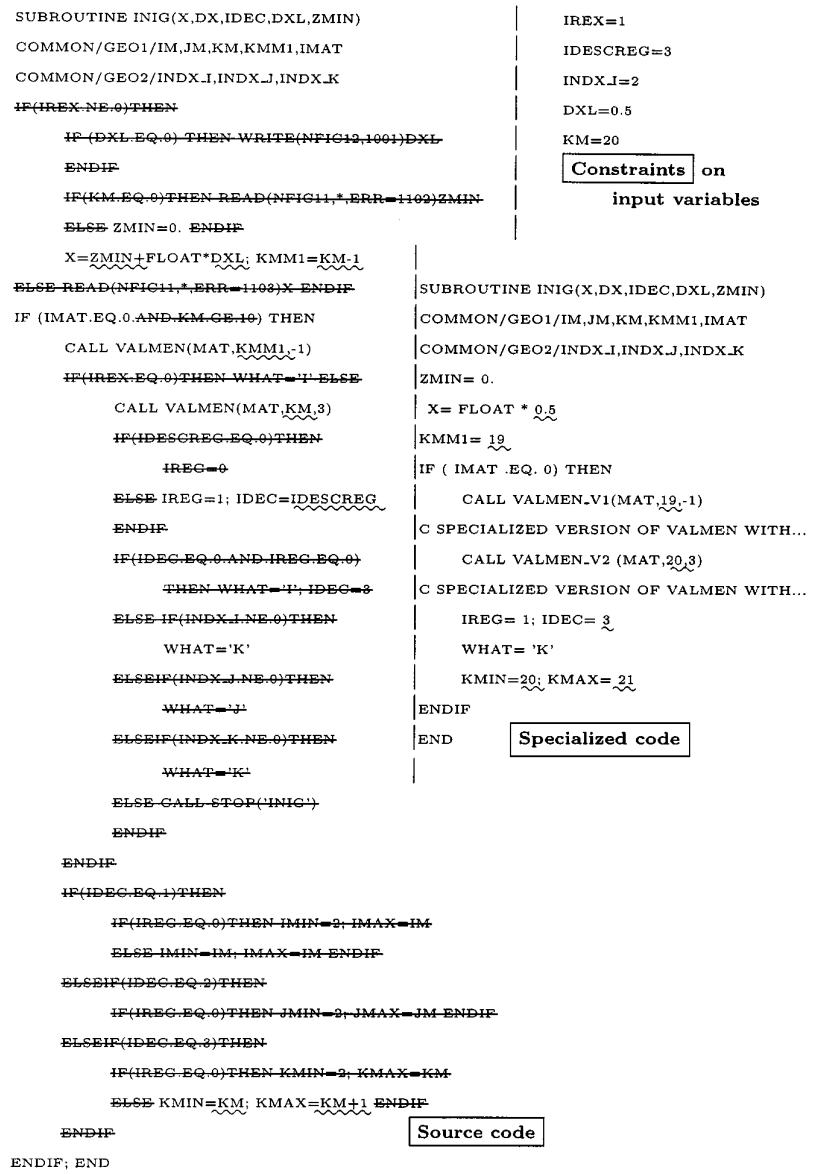


Figure 1. An example of program specialization.

expressions are also simplified. In source and specialized codes, simplified expressions are underwaved. Known values of variables are propagated in called procedures. Called procedures have been replaced by their specialized versions and a comment recalls the name of the called procedure and its initial known values. Other information is computed and displayed during the partial evaluation (e.g. final values of some variables). It is not shown in figure 1 to simplify the presentation of the figure.

1.2. Technical overview of our interprocedural alias analysis

Our partial evaluator implements a flow-sensitive and context-sensitive interprocedural alias analysis (Emami et al., 1994; Hasti and Horwitz, 1998; Landi and Ryder, 1992; Liang and Harrold, 1999; Wilson and Lam, 1995). In a flow-sensitive analysis, aliasing information depends on control flow: the analysis takes into account the order in which statements are executed. In a context-sensitive analysis, aliasing information is differentiated among call sites: the analysis takes into account the fact that a function must return to the site of the most recent call.

Our analysis computes *must* alias information (i.e. alias information that must occur on all paths through the flowgraph). For each pointer and each program point, our analysis computes a conservative approximation of the memory locations to which that pointer must point.

Flow-sensitive analyses generally take more time and space than flow-insensitive analyses; however, the results are usually more precise (Hasti and Horwitz, 1998). Our analysis is more sensitive than other alias analysis that are more efficient (Andersen, 1994; Carini and Hind, 1995; Sagiv et al., 1997; Steensgaard, 1996), but our analysis does not take prohibitive time or space. The reason is that our partial evaluation propagates only equalities between variables and values. This means that when an if statement or a loop is analyzed, some information is lost at the end of the analysis of the statement:

- when two branches of an if statement are analyzed, the analysis propagates only the equalities between variables and values that hold in both branches,
- in a similar way, when a loop is analyzed, only the equalities between variables and values that belong to the invariant of the loop are propagated outside the loop.

Our analysis does not support recursive calls (the application programs we have analyzed are not recursive) but it handles return constants. The framework of our analysis is similar to the more general one described by Chase and Zadeck in Chase and Zadeck (1990): for each procedure, information that describes the effects of that procedure are propagated through the call graph. This graph represents the structure of the analyzed program.

1.3. A formalism for specifying program transformations

We use natural semantics rules to specify and implement our relations. These are inference rules that are made of sequents defining inductively our relations. The rules operate on

abstract syntax trees. Each rule expresses how to deduce sequents (the denominator of the rule) from other sequents (the numerator of the rule) (Kahn, 1987). While natural semantics rules provide a simple means for specifying our program transformation, they use very simple data structures for representing programs and other information, and they use only very simple formulas and inference rules for expressing program transformations. Hannan (1993) explains that this simplicity has the advantage of yielding straightforward and efficient implementations of the rules, but it also has the obvious disadvantage of yielding specifications that contain primitive encoding of program transformations.

In our interprocedural alias, we need to specify side-effects on global variables and pointers, and side-effects accomplished through parameter passing. Thus, we have extended the natural semantics formalism. To simplify the presentation of the rules:

- We have used various set and relational operators to specify our program transformations. The computations using these operators are not defined in the rules themselves, but before the rules in a section called "definitions" (e.g. see figure 6).
- The links between nodes of abstract syntax trees (representing pieces of programs) and all other data associated with specifications have been modeled in object diagrams. Instances of the objects are data appearing in the rules, and arrows between objects are functions applied to data appearing in the rules (e.g. see figure 4). Only the instances of some objects (those that consist of other objects) of the diagram occur in the rules. The instances of other objects and the arrows between objects are only defined in the diagrams.

Thus, in this paper, a specification consists of definitions of data, a natural semantics rule (with data defined in the definition part) and an object diagram showing composition links between the data of the rule. As there is no formal description of the semantics of Fortran 90, we have first specified a dynamic semantics of Fortran 90. We have then specified our partial evaluation and this specification is based on the specification of the dynamic semantics.

1.4. Overview of the paper

The aim of this paper is to show how we have specified, proven and implemented our partial evaluator, and especially our interprocedural alias analysis. Compared to our previous work (Blazy and Facon, 1994; 1995; 1996):

- Our program analyses take into account an interprocedural alias analysis.
- The definition of our partial evaluation is based on a definition of the dynamic semantics of Fortran 90.
- We have extended the natural semantics formalism to present higher-level specifications.
- We have implemented a graphical interface.

The rest of this paper is structured as follows. First, Section 2 recalls some concepts of Fortran 90 and gives some definitions that are useful to understand our specifications. Then, Section 3 specifies the dynamic semantics of call statements and pointers. Section 4

specifies the partial evaluation of call statements. It shows how the propagation relation has been derived from the dynamic semantics relation, and it presents the general framework for proving the correctness of the partial evaluation. Section 5 is devoted to the implementation of our tool.

2. Background and notations

This section gives a brief overview of the features of Fortran 90: procedures, common blocks, structures, pointers and targets. Then, it defines set and relational opertors that will be used in the specifications.

2.1. Fortran 90

The most interesting aspects of Fortran 90 for partial evaluation are the treatment of parameters, variables and structures, when they are passed between procedures (i.e. subroutines or functions). In Fortran 90, procedures cannot be nested. Parameters may be modified in procedures. Certain side effects are prohibited by the Fortran standard (e.g. the statement `X = F(I) + I` is not allowed if the function `F` changes the value of `I`). Other side effects are permitted, but the result of using them is not defined (i.e. it is processor dependent). Some Fortran implementations use a call-by-value-result semantics. Other implementations use a call-by-reference semantics (Aho et al., 1988). Both semantics are allowed by the Fortran standard, and we have chosen to specify a call-by-value-result semantics.

Fortran 90 has no global storage other than variables in common blocks. Common blocks represent contiguous areas of memory. The variables of common blocks are shared across procedures. Common blocks are unusual for modern programming languages in that they associate variables by location, rather than by name. Although the names of the common blocks themselves are global, none of the variables within the common blocks are global. Common blocks may also be inherited across procedure calls. Given a common block *CB* declared in a procedure P, the rule of dynamic scoping for *CB* states that *CB* can be referred to, not only in P, but also in any procedure called from within P, even if *CB* is not declared in such a procedure. If a common block is neither declared in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block are undefined.

A structure consists of a list of fields, each of some particular type. The field types may include pointers to structures of the type being defined, of a type previously defined, or of a type yet to be defined. A pointer may point to any object that has the `TARGET` attribute, which may be any dynamically allocated object or a named object declared with that attribute. When the statement `NULLIFY(`p`)` is executed, the pointer p becomes `NULL`. In Fortran 90, a pointer should be thought of as a variable aliased to another data object where the data is actually stored—the target (ANSI, 1992; Muchnick, 1997). There is no pointer arithmetic in Fortran 90. There is no notation for representing pointed variables (dereferencing is automatic in Fortran 90). We will then use a C-notation when needed for clarity (e.g. *$*(*p.next)$* in figures 2 and 3).

```
TYPE node
   REAL :: ident                  ! data field
   TYPE(node), POINTER :: next    ! pointer field
END TYPE node
 ...
 q => p%next    ! q points to (i.e. is an alias of) *(*p.next)
 p%ident = 3.4  ! the value 3.4 is assigned to the field ident of p
 q%ident = 6.2
```

Figure 2. An example of Fortran 90 code.

Figure 2 shows a Fortran 90 program that defines a new type called `node`. The `node` type contains two fields: `ident` containing the value associated with the node, and `next` pointing to the next element of the list. Once two variables p and q of type pointer to `node` have been declared, the values 3.4 and 6.2 are inserted in the list of `nodes` in that order. The insertion in the list modifies the three variables q, $p\%ident$ and $q\%ident$.

The syntax of identifiers is specified by the abstract syntax of figure 3. An identifier is either a simple identifier (e.g. v), or a compound left-hand side (e.g. ${}^{*}p.next$), or a pointer dereference (e.g. ${}^{*}p$, ${}^{*}({}^{*}p.next)$). The set of simple variables identifiers of a procedure is denoted by *VarName*. The set of left-hand sides is denoted by *Lhs*. Among left-hand sides are simple variables. Thus, $VarName \subseteq Lhs$. The example in this figure shows the connection between some concrete Fortran 90 variables and the corresponding abstract syntax notations.

Abstract syntax

$$VarName \subseteq Lhs$$

$$\mathbf{deref} : Lhs \to Lhs$$
$$\mathbf{field\_lhs} : Lhs \times VarName \to Lhs$$

Examples

concrete syntax	abstract syntax
v	$VarName(v)$
$person\%address\%town$	$\mathbf{field\_lhs}(\mathbf{field\_lhs}(person, address), town)$
$*((*p).next)$	$\mathbf{deref}(\mathbf{field\_lhs}(\mathbf{deref}(p), next))$

Figure 3. Abstract syntax rules and examples of links with concrete syntax.

2.2. *Definitions*

Before going into the details of how to specify the dynamic semantics and the partial evaluation, we define in this section some notation, especially set operators, that we use in our specifications. We first introduce useful set operators, similar to those defined in the formal specification languages B (Abrial, 1996) and VDM (Jones, 1990): mainly inverse ($^{-1}$), domain (**dom**), range (**ran**), union ($\cup$), override ($\dagger$), restriction ($\rhd$, $\lhd$ and $\lhd\!\!\!\!-$), relation composition ($;$ [1]) and direct product ($\otimes$). In the following definitions s denotes a set, r, p and q denote binary relations, m and n denote maps (specific binary relations where each element is related to at most one element). A binary relation is a set of pairs. Thus, classical set operators such as union can also be applied to binary relations. For each operator that we define, Table 1 shows through an example of how the operator can be used.

- $r^{-1} = \{x \mapsto y \mid y \mapsto x \in r\}$
- $\mathbf{dom}(r) = \{x \mid x \mapsto y \in r\}$
- $\mathbf{ran}(r) = \{y \mid x \mapsto y \in r\}$
- $r \dagger p = \{x \mapsto y \mid x \mapsto y \in p \bigvee (x \mapsto y \in r \bigwedge x \notin \mathbf{dom}(p))\}$
- $r \rhd s = \{x \mapsto y \in r \mid y \in s\}$
- $s \lhd r = \{x \mapsto y \in r \mid x \in s\}$
- $s \lhd\!\!\!\!- r = \{x \mapsto y \in r \mid x \notin s\}$
- $r; p = \{x \mapsto z \mid \exists y \cdot x \mapsto y \in r \bigwedge y \mapsto z \in p\}$
- $r \otimes p = \{x \mapsto (y, z) \mid x \mapsto y \in r \bigwedge x \mapsto \dot{z} \in p\}$

When r and p are functions, their direct product is especially interesting when it is an injective function. In this case, when a pair of the form $x \mapsto (y,\ z)$ belongs to $r \otimes p$ then $(y,\ z)$ determines x uniquely, and x may be written $(r \otimes p)^{-1}(y,\ z)$. The $\otimes$ relation generalizes naturally to more arguments.

Furthermore, we have defined an operator to bind variable names of a common block to their corresponding values. Given s a set of pairs of maps (e.g. s is of the form

Table 1. Examples of notations.

Operator	Example
$^{-1}$	$\{1 \mapsto a, 3 \mapsto u\}^{-1} = \{a \mapsto 1, u \mapsto 3\}$
dom	$\mathbf{dom}(\{1 \mapsto a, 3 \mapsto u\}) = \{1, 3\}$
ran	$\mathbf{ran}(\{1 \mapsto a, 3 \mapsto u\}) = \{a, u\}$
$\dagger$	$\{1 \mapsto a, 3 \mapsto u\} \dagger \{1, \mapsto b, 2 \mapsto d\} = \{1 \mapsto b, 2 \mapsto d, 3 \mapsto u\}$
$\rhd$	$\{1 \mapsto a, 3 \mapsto u\} \rhd \{a, b\} = \{1 \mapsto a\}$
$\lhd$	$\{1, 2\} \lhd \{1 \mapsto a, 3 \mapsto u\} = \{1, \mapsto a\}$
$\lhd\!\!\!\!-$	$\{1, 2\} \lhd\!\!\!\!- \{1 \mapsto a, 3 \mapsto u\} = \{3, \mapsto u\}$
;	$\{1 \mapsto a, 3 \mapsto u\}; \{a, \mapsto 67, b \mapsto 26\} = \{1 \mapsto 67\}$
$\otimes$	$\{1 \mapsto a, 3 \mapsto u\} \otimes \{1 \mapsto b, 2 \mapsto d, 3 \mapsto z\} = \{1 \mapsto (a, b), 3 \mapsto (u, z)\}$

$\{m_1 \mapsto n_1, m_2 \mapsto n_2, m_3 \mapsto n_3, \ldots\}$, where $m_1, n_1, m_2, n_2, m_3, n_3, \ldots$ are maps), we define:

$$\mathbf{Corres}(s) = \bigcup \{m^{-1}; n \mid m \mapsto n \in s\}.$$

Variables of common blocks are shared among procedures (their values are inherited in each called procedure) but their names may change in each procedure. Thus, each pair $m \mapsto n$ of s corresponds to a common block. m and n map integers (the position in the list of declared variables of the common block) to respectively variable names and values. $m^{-1}; n$ is then a map from variable names to values and **Corres**(s) is the union of all such maps.

In the following example, the declaration of two common blocks `B` and `C` is represented by the map *ComDecl*. At the current program point, the values of variables belonging to common blocks are given by the map *ComVal*. *ComVal* states that:

- the value of the first variable of `B` is `0` and the value of the other variable of `B` is unknown (this variable is not represented in the map *ComVal*),
- the value of the first variable of `C` is 0.5, the value of the second variable of `C` is unknown, and the value of the third variable of `C` is 6.2.

In the current procedure, the variables of `B` are `t` and `u`, and the variables of `C` are `x`, `y` and `z`. Thus, at the current program point, the map from these variables to their values is **Corres**(s).

EXAMPLE.
```
COMMON B / t, u
COMMON C / x, y, z
```

$$\begin{aligned} \mathit{ComDecl} &= \{B \mapsto \{1 \mapsto t, 2 \mapsto u\}, C \mapsto \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\}\} \\ \mathit{ComVal} &= \{B \mapsto \{1 \mapsto 0\}, C \mapsto \{1 \mapsto 0.5, 3 \mapsto 6.2\}\} \\ \text{If } s &= \mathit{ComDecl}^{-1}; \ \mathit{ComVal} \quad \text{then} \end{aligned}$$

$$s = \{\{1 \mapsto t, 2 \mapsto u\} \mapsto \{1 \mapsto 0\}, \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\} \mapsto \{1 \mapsto 0.5, 3 \mapsto 6.2\}\}$$
$$\text{and} \quad \mathbf{Corres}(s) = \{t \mapsto 0, x \mapsto 0.5, z \mapsto 6.2\}$$

3. Specification of the dynamic semantics

This section details the specification of the dynamic semantics of two kinds of statements: call statements and assignments between pointers. The dynamic semantics of assignments between targets and pointers is specified in a similar way, but is not presented in this paper. Each specification is presented in two sections: the first section details an object diagram and the second section gives definitions and natural semantics rules. These rules use data that are defined either in the object diagram or in the definitions.

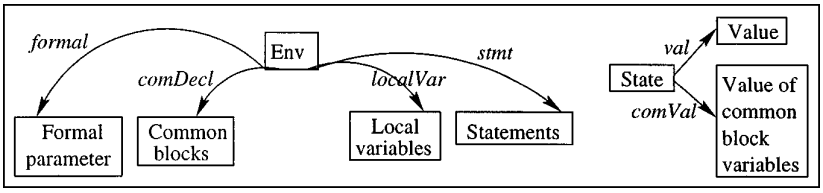


Figure 4. The state and the environment of a procedure.

3.1. Object diagram representing programs

The dynamic semantics specifies how procedures are executed. To execute a call statement we execute the body of the called procedure. This means that we have to keep track of the association of procedures 1) with their bodies and also 2) with the values of their variables. Thus, in figure 4, we define two objects for representing a Fortran 90 procedure:

1. an environment (called Env in figure 4), that represents what does not vary during the analysis of the procedure (formal parameters, common block variables that have a scope in the current procedure, local variables and statements),
2. a state (called State in figure 4) modeling relations between variables and values at the current program point. When a procedure is analyzed, each analysis of a statement updates the state of the procedure. An object State (called S) consists of two objects: *val*(S) and *comVal*(S). *val*(S) maps variables to values and *comVal*(S) maps common blocks to the known values of their variables. Since in common blocks, values are shared between two variables simply by the fact that they occupy corresponding positions within the same common block, these values are modeled differently from the values of other variables. In figure 4 these values are represented by the object called "Value of common block variables" that is accessed from State through the *comVal* function.

The diagram of figure 5 extends the diagram of figure 4, to model the whole data that represent procedures. In figure 5, the object called Procedure denotes a called procedure SP

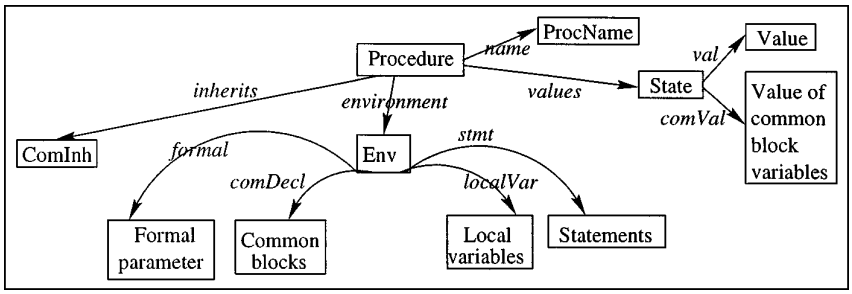


Figure 5. Object diagram showing data propagated during an interprocedural analysis.

at a given program point. From SP, we have access to its environment *environment*(SP), its state *values*(SP), but also to its inherited common blocks *inherits*(SP) and its name *name*(SP). This name belongs to ProcName, that denotes the set of procedure names that are used in the current application program.

An instance of the diagram of figure 5 represents information for all procedures that are called from a main program (or from one of its called procedures). It shows the bindings between procedure names and their corresponding statements. The objects of the instance are never modified by the sequents. Thus, the instance is an implicit parameter of the dynamic semantics and partial evaluation systems, to simplify the presentation of the rules.

3.2. Natural semantics rule for the dynamic semantics of a call statement

The dynamic semantics of Fortran 90 is formalized by the *sem* system of inference rules, that generates sequents of the form $\mathsf{E}, \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \mathsf{I} : \mathsf{S}'$, meaning that given an environment E, a state S and inherited common blocks Cl, the execution of statement I leads to the values given by the state S′. E, S and Cl are the hypotheses of the sequent. Figure 6 shows the dynamic semantics rule for a call statement `CALL SP (LParam)`. SP is the name of the called procedure and LParam is a map from positions to the actual parameters of SP at the current program point. The rule specifies a call-by-value-result dynamic semantics.

The beginning of figure 6 illustrates through an example which data are propagated during the execution of statements with respect to the dynamic semantics rule of the figure. In this example, these statements are located at the program points α, β, γ and δ. The execution of the call statement starts with a forward propagation through the call graph (from program point α to program point γ) followed by a backward propagation (from program point δ to program point β). The forward propagation aims at giving new values to formal parameters and common blocks variables of the called procedure SP. These values are *val*(S) (values of variables) and *comVal*(S) (values of common blocks). The backward propagation aims at giving new values to the actual parameters of the called procedure and to the variables belonging to common blocks of the caller that have a scope in the called procedure.

In the definitions part of the figure, some definitions are factorized to simplify the presentation of the dynamic semantics rule. They introduce some macros used in specifying the dynamic semantics rule. In the definitions, the $\underline{\Delta}$ symbol means "by definition". The first definition defines EnvSP, the environment of the procedure named SP. $name^{-1}$ yields an object Procedure from an object ProcName (see figure 5). Thus, since SP is a procedure name, $name^{-1}$(SP) represents the procedure whose name is SP and EnvSP is defined as the environment of this procedure.

In the rest of Section 3.2, we describe each of the macros in more detail. To describe the macros, we follow the execution order of the statements. The first five macros compute the values that are transmitted to the called procedure SP. The general idea is that the macros update the initial state S and inherited common blocks C so that when executing the statements of EnvSP, the data declared in SP will be available and the side-effects will be taken into account. The macros detailed in the last subsection compute the values that SP transmits back to the call site that invoked it.

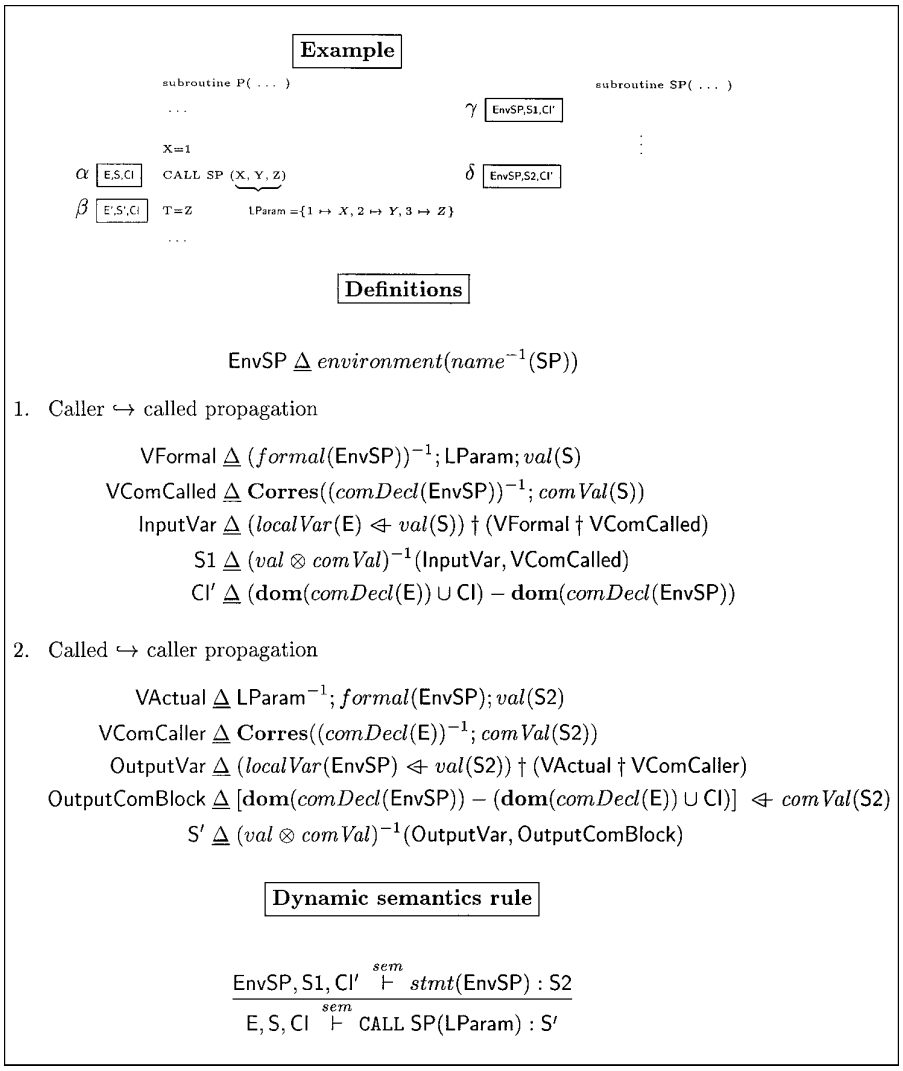


Figure 6. Dynamic semantics of a call statement.

3.2.1. Computation of VFormal (values of formal parameters at program point γ). Formal (resp. actual) parameters are represented by maps between positions and names (resp. values) of the parameters. In figure 6, these maps are respectively *formal*(EnvSP) and LParam.

- (*formal*(EnvSP))$^{-1}$ maps the names of the formal parameters of SP to their positions,
- LParam; *val*(S) is the result of the evaluation of the actual parameters at the program point α. It maps the names of the actual parameters to their current values.

- Thus, $(formal(\mathsf{EnvSP}))^{-1}$; LParam; $val(\mathsf{S})$ maps the names of the formal parameters of SP to their initial values, i.e. the values of actual parameters at the program point α.

3.2.2. Computation of VComCalled (values of common blocks at program point γ). Variables belonging to a common block are represented by a map between positions in the common block and variable names. The values of the variables that belong to the common blocks of the calling procedure are also transmitted to the corresponding variables (they share a same position in the common blocks). Thus,

$$\mathsf{VComCalled} \underline{\triangle} \mathbf{Corres}((comDecl(\mathsf{EnvSP}))^{-1}; comVal(\mathsf{S}))$$

is a list of pairs (variable of a common block declared in SP, value) (see the example of Section 2.2), and it is an instance of the object called "Value of common blocks variables" in figure 5.

3.2.3. Computation of InputVar (values of variables at program point γ). The variables of SP (and their initial values) are:

- variables of the calling procedure P that have a scope in SP. These are the current static variables $val(\mathsf{S})$.
- formal parameters and variables of common blocks that are declared in SP (i.e. VFormal†VComCalled): in SP, if a formal parameter is also declared in a common block, then its value is the value given by the common block.

The restriction between the two maps VFormal†VComCalled and $localVar(\mathsf{E}) \lhd val(\mathsf{S})$ models scope rules between procedures. The resulting map InputVar belongs to the object called Value in figure 5.

3.2.4. Computation of S1 (known values at program point γ). $val \otimes comVal$ is an injective function:

- Any state State represents at least one program point, where values of some variables are known. These values are precisely given by the pair of maps $(val \otimes comVal)(\mathsf{State})$. Thus, $val \otimes comVal$ is a total function.
- Given two states S1 and S2, $(val \otimes comVal)(\mathsf{S1}) = (val \otimes comVal)(\mathsf{S2})$ means by definition of a pair that $val(\mathsf{S1}) = val(\mathsf{S2})$ and $comVal(\mathsf{S1}) = comVal(\mathsf{S2})$. This means that all the values of static variables are the same, and that the static variables are also the same in S1 and S2. Thus, S1 and S2 denote the same state.

We can then write $(val \otimes comVal)^{-1}(\mathsf{InputVar}, \mathsf{VComCalled})$ to denote a State object for SP, as explained previously (see Section 2.2). We call this state S1.

3.2.5. Computation of Cl′ (names of inherited common blocks at program point γ). The common blocks Cl′ that are inherited by SP are those that have a scope in its caller except those that are also declared in SP (these are **dom**$(comDecl(\mathsf{EnvSP}))$, where the domain of the map $comDecl(\mathsf{EnvSP})$ yields the names of common blocks). The common blocks

that have a scope in the caller are its declared common blocks **dom**(*comDecl*(E)) and its inherited common blocks Cl.

3.2.6. Premise of the inference rule. Once S1 and Cl′ have been computed, values have been transmitted to SP and the forward propagation ends. The current program point is γ. Then, the premise of the dynamic semantics rule is triggered. As indicated by this premise, the statements of SP are executed given the environment of SP (EnvSP), its state (S1) and inherited common blocks (Cl′), yielding a new State object S2 for SP. S2 represents the new values (for common blocks and parameters of SP) that need to be transmitted to the calling procedure.

3.2.7. Computations of the backward propagation (second part of definitions in figure 6). Once the premise of the rule has been triggered, the analysis has reached program point δ. The known values are then transmitted back to the calling procedure: the maps LParam and *comDecl*(E) (from actual parameters and common blocks variables to their values) are updated, yielding the final state S′ of the caller. Similarly to VFormal in the forward propagation,

$$\mathsf{VActual} \triangleq \mathsf{LParam}^{-1}; \mathit{formal}(\mathsf{EnvSP}); \mathit{val}(\mathsf{S2})$$

is a list of pairs (actual parameter, value of formal parameter). The definition of VComCaller (resp. OutputVar) is very close to the definition of VComCalled (resp. InputVar).

SP has inherited the values of common blocks from P. But, the values of some common blocks of SP are not backward propagated in P. These are the common blocks declared in SP that do not have a scope in P. Thus, the definition of S′ differs slightly from the definition of S1. The values OutputComBlock of common blocks at program point β are the values of common blocks at program point δ (i.e. *comVal*(S2)), except for the common blocks that we call CB that are declared in SP and that do not have a scope in P. This means that OutputComBlock is defined as follows:

$$\mathsf{OutputComBlock} \triangleq \mathsf{CB} \lhd \mathit{comVal}(\mathsf{S2})$$

Among the common blocks declared in SP (i.e. **dom**(*comDecl*(EnvSP))), CB represents the common blocks that are neither declared in P, nor inherited by P (from one of its callers). Thus, CB is defined as follows:

$$\mathsf{CB} \triangleq \mathbf{dom}(\mathit{comDecl}(\mathsf{EnvSP})) - (\mathbf{dom}(\mathit{comDecl}(\mathsf{E})) \cup \mathsf{Cl})$$

and the values OutputComBlock of common blocks at program point β are:

$$[\mathbf{dom}(\mathit{comDecl}(\mathsf{EnvSP})) - \mathbf{dom}(\mathit{comDecl}(\mathsf{E})) \cup \mathsf{Cl})] \lhd\!\!- \mathit{comVal}(\mathsf{S2}).$$

3.3. Object diagram representing pointers

As Nielson and Nielson (1992), we define two kinds of maps for representing pointers: a variable environment that associates a location with each variable, and a store that associates a value with each location. The variables are represented by locations in stores. The set of

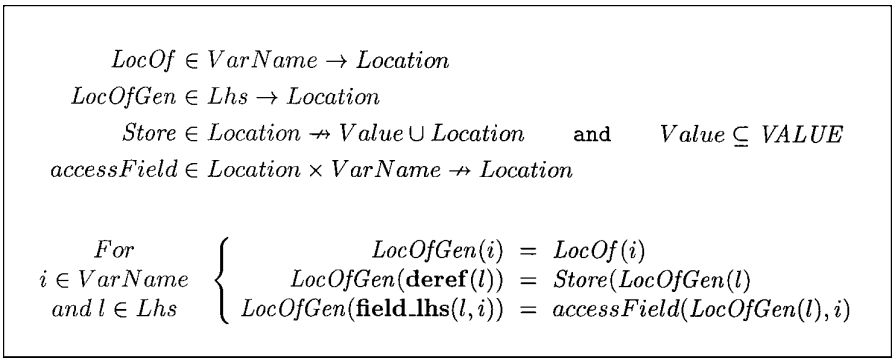


Figure 7. Dynamic semantics of some variables.

values of variables that have a scope in a given procedure is denoted by *Value*. It is a subset of *VALUE*, the syntactic category representing values of variables. The set of locations of a given procedure is denoted by *Location*. The dynamic semantics of pointers is modeled by the following functions, which are formally defined in figure 7:

- *LocOf* maps (simple) identifiers to their locations.
- The map *LocOfGen* extends the *LocOf* map to left-hand sides and dereferences. The location of a pointed record is the location of its first field.
- The map *Store* represents the aliases between variables and their values. These values belong either to the set *Value* or to the set *Location*. For instance:
  - If the value of a variable i is 3.4, then in the store this information is modeled by a pair (location of i, 3.4), where 3.4 belongs to *Value*.
  - If a pointer p points to a target $^*q$, then in the store this information is modeled by a pair (location of p, location of q).

  If a pointer is `NULL`, it does not point to anything, and thus it is not represented in the *Store* function. Thus, *Store* is a partial function.
- given the location of a record r and a field f, *accessField* yields the location of $r.f$. This is a partial function since only record names with their corresponding fields may have a location.

In figure 7, the operators **deref** and **field_lhs** belong to the Fortran abstract syntax (see figure 3).

The function *val* mapping variables to their known values (see figures 4 and 5) has been split into two maps *LocOfGen* and *Store*. It is now defined as follows:

$$val \triangleq LocOfGen; Store \triangleright VALUE$$

This means that to determine the value of a variable i we shall first determine the location $l = LocOfGen(i)$ associated with i, and then determine the value *Store*(l) associated with

the location l. As *Store*(l) is either a location or a value, the map *LocOfGen*; *Store* is restricted to possible values.

The information about the locations to which a pointer points is modeled by *pointsTo*. *pointsTo* is a map from pointers to their targets, that is defined as follows:

$$pointsTo \triangleq LocOfGen; Store; LocOfGen^{-1}$$

Example. If we consider a program point where a pointer p points to a target $^*q$, a pointer r points to a memory cell whose location is denoted by *loc* and there is no other pointer, then at this program point, we will have $pointsTo = \{p \mapsto \mathbf{deref}(q), r \mapsto loc\}$.

Figure 8 represents in diagrammatic form the linked list created by the statements of figure 2. It also shows the dynamic semantics of the corresponding statements. All pointer chaining is resolved before the last two assignments, so any node can be referred to directly by its location. Each node has been dynamically allocated. Thus, each node has a unique location, as shown in the map *LocOfGen*. The definition of this map is illustrated in the second part of figure 8, where the location of *p%next%ident* is computed. The last part of figure 8 details the maps *val* and *pointsTo*:

- *val* states that the value of $^*p$ is 3.4 and the value of **((* p).next)* is 6.2.
- *pointsTo* states that p points to $^*p$ and q points to **((* p).next)*.

Figure 9 models all of the data that are propagated during the execution of any statement. It extends figure 5 by showing functions modeling pointer variables. The maps *LocOfGen* and *Store* are denoted by two objects that are accessed from the object called State. Values of variables are now given by the *store* function. Thus, the object called State may be considered as a triplet *state* (set of locations, set of stores, set of values for variables of common blocks), where *state* denotes a constructor of occurrences of this object.

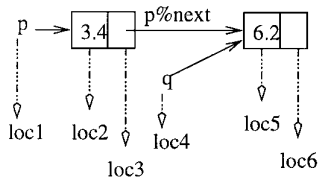
3.4. Natural semantics rule for the dynamic semantics of assignments between pointers

Figure 10 shows the dynamic semantics of a pointer assignment $p \Rightarrow q$, given two pointers p and q, an environment E, a state S and inherited common blocks Cl (same hypotheses as in the rule for a call statement). The execution of the assignment modifies only locations and stores. This means that it updates only the current state S. The current locations and stores are respectively $\mathsf{L} \triangleq locOfGen(\mathsf{S})$ and $\mathsf{ST} \triangleq store(\mathsf{S})$. The dynamic semantics expresses that L and $\mathsf{ST}$are updated by the alias introduced by that assignment.

Before the execution of the assignment $p \Rightarrow q$, one of the following three situations arises:

1. q is not `NULL` and points to a target,
2. q is `NULL`,
3. q is not `NULL` and does not point to any target.

Representation of a list



$$
\begin{aligned}
LocOf &= \{p \mapsto loc_1, q \mapsto loc_4\} \\
LocOfGen &= \{\mathbf{deref}(p) \mapsto loc_2, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto loc_5\} \cup LocOf \\
Store &= \{loc_1 \mapsto loc_2, loc_3 \mapsto loc_5, loc_4 \mapsto loc_5, loc_2 \mapsto 3.4, loc_5 \mapsto 6.2\} \\
accessField &= \{(loc_2, next) \mapsto loc_3, (loc_2, ident) \mapsto loc_2, (loc_5, ident) \mapsto loc_5, \\
&\quad (loc_5, next) \mapsto loc_6\}
\end{aligned}
$$

Location of the cell $p\%next\%ident$

The abstract syntax of $p\%next\%ident$ is:
$\mathbf{fieldLhs}(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)), ident)$.

$$
\begin{aligned}
&LocOfGen(\mathbf{fieldLhs}(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)), ident)) = \\
&accessField(LocOfGen(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next))), ident) = \\
&accessField(Store[LocOfGen(\mathbf{fieldLhs}(\mathbf{deref}(p), next))], ident) = \\
&accessField(Store[accessField(LocOfGen(\mathbf{deref}(p), next))], ident) = \\
&accessField(Store[accessField(Store(LocOf(p)), next)], ident) \\
&accessField(Store[accessField(Store(loc_1), next)], ident) = \\
&accessField(Store(loc_3), ident) = loc_5
\end{aligned}
$$

Maps *val* and *pointsTo*

$$
\begin{aligned}
LocOfGen; Store &= \{\mathbf{deref}(p) \mapsto 3.4, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto 6.2, \\
&\quad p \mapsto loc_2, q \mapsto loc_5\} \\
val &= \{\mathbf{deref}(p) \mapsto 3.4, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto 6.2\} \\
pointsTo &= \{p \mapsto \mathbf{deref}(p), q \mapsto \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next))\}
\end{aligned}
$$

Figure 8. An example of linked list.

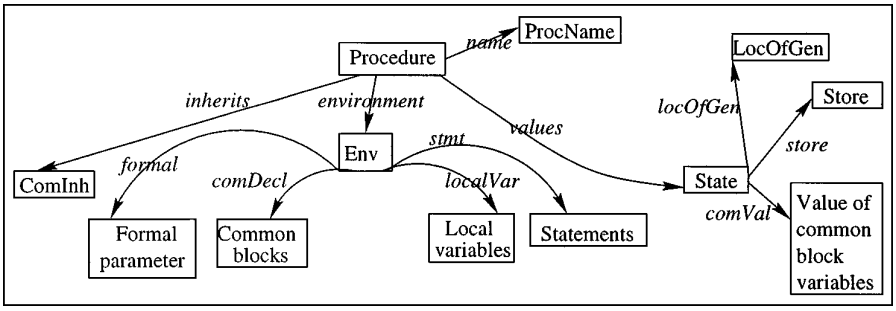


Figure 9. Object diagram showing data propagated during the execution of a procedure.

These three situations are modeled by the three dynamic semantics rules of figure 10. The first rule is triggered when q points to a target ${}^*q$. This happens only if $q \in \mathbf{dom}(pointsTo(\mathsf{S}))$. In this case, when p is assigned to the value of q:

- p points to ${}^*q$. The value in the store at the location of p (i.e. $\mathsf{ST}(\mathsf{L}(p))$) becomes the location of q (i.e. $\mathsf{L}(q)$), yielding a new map of locations $\mathsf{L1}$.
- the location of ${}^*p$ becomes the location of ${}^*q$, $\mathsf{ST}(\mathsf{L}(q))$, yielding a new map of stores $\mathsf{ST1}$.

A new state $\mathsf{S1}$ is then created from the three components $\mathsf{L1}$, $\mathsf{ST1}$ and $comVal(S)$. This creation is very close to the creation of the states $\mathsf{S1}$ and $\mathsf{S'}$ in the dynamic semantics of call statements (Section 3.2).

In the second rule, q has the NULL value. This means that $q \notin \mathbf{dom}(store)(\mathsf{S})$ (see Section 3.3) and thus by definition of *pointsTo*, $q \notin \mathbf{dom}(points\ To(\mathsf{S}))$. In this case, ${}^*p$ has no location anymore. ST becomes $\{L(p)\} \lhd \mathsf{ST}$. A new state S2 is created from this new store and from the two other components that have not changed.

In the third rule, there is no location pointed by q. Then, the maps of locations and stores are restricted in the following way:

- p does not point to any location. L becomes $\{\mathbf{deref}(p)\} \lhd \mathsf{L}$.
- ${}^*p$ has no location anymore. As in the second rule, ST becomes $\{\mathsf{L}(p)\} \rhd \mathsf{ST}$.

4. Specification of the partial evaluation

This section shows through an example how the specification of the partial evaluation is derived from the specification of the dynamic semantics. There are two main differences between the dynamic semantics and the partial evaluation. First, in the partial evaluation, some variables have unknown values. Second, to improve the partial evaluation, specialized procedures are reused as often as possible. The example presented in this section concerns the partial evaluation of call statements.

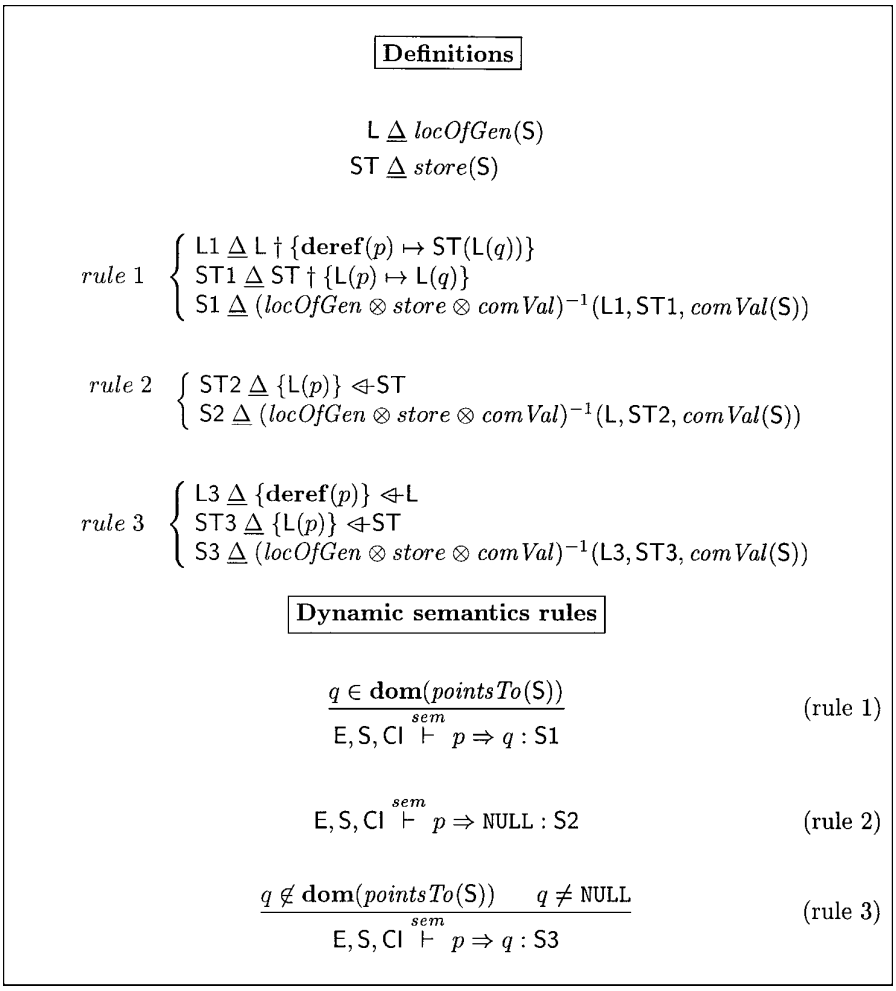


Figure 10. Dynamic semantics of a pointer assignment.

As in the dynamic semantics, the specification is presented in two parts: the first part details the object diagram representing specialized versions (i.e. procedures that have already been specialized), and the second part gives the natural semantics rules that specify the partial evaluation of call statements. The section ends with a sketch of a proof of correctness of the partial evaluation.

4.1. Object diagram representing specialized versions

Specialization proceeds depth-first in the call-graph to preserve the order of side-effects. Thus, the specialization of a call statement first runs the specializer on the called

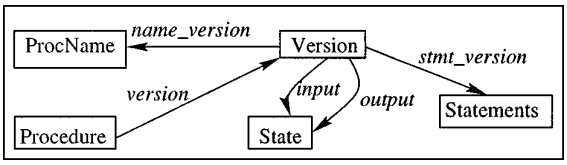


Figure 11. Object diagram modeling specialized versions.

procedure SP. This yields a specialized version of SP and the call statement is replaced by a call to this specialized version. A procedure is specialized with respect to specific values of some of its input data (its input static data). At the end of its specialization, the known values of variables belong to its output static data, and a new name is given to the new specialized version (if any).

The diagram of figure 11 models specialized versions. A procedure SP belongs to the object called Procedure. SP represents the code of the whole procedure (its declarations and its statements). The specialized versions of SP belong to the object called Version. They are represented by the set *version*(SP). A specialized version v of SP consists of a name for this version (*name_version*(v)), input static data (*input*(v)), output static data (*output*(v)) and statements (*stmt_version*(v)). This is equivalent to saying that a version is represented by a quintuplet (name of original procedure, version name, input data, output data, statements). The version and its corresponding procedure have the same arity. Thus, the procedure and the version have the same formal parameters. The name of the version is a name that could be the name of a procedure.

To improve the specialization, specialized versions of procedures are reused. Thus, given a set of specialized procedures, when a call to a procedure SP is encountered in the current procedure (e.g. `CALL SP(x, y, z)`), if the set of input static data of SP and their values (e.g. `x` evaluates to `1`, `y` evaluates to `2` and the value of `z` is unknown):

- is the same as those of a previous call, then the corresponding version is directly reused,
- strictly includes those of a previous call (e.g. `CALL SP(a,b,c)` where a evaluates to `1` and the values of `b` and `c` are unknown), then the corresponding version is specialized yielding a new version that is added to the already specialized versions. If several versions match, the following selections are successively made:
  - most specialized versions, that is the versions with the largest set of input static data,
  - shortest version among most specialized versions.

The partial evaluation of a program is an analysis that propagates:

- the same data as the dynamic semantics (environment, state and inherited common blocks),
- specialized versions of already specialized procedures.

This means that in the sequents for partial evaluation, the data that are written at the left of the turnstile are an environment (e.g. E), a state (e.g. S), some inherited common blocks

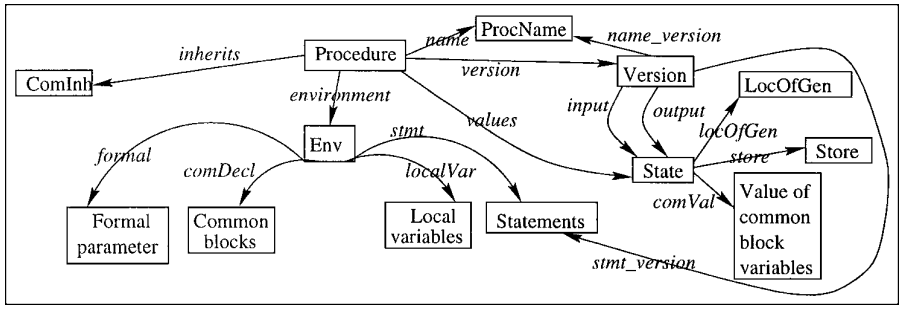


Figure 12. Object diagram showing data propagated during partial evaluation.

(e.g. Cl), and also a set (e.g. V) of versions. These versions are procedures that have been already called and then specialized.

Figure 12 shows data that are propagated during the partial evaluation. It extends figures 9 and 11. As in figure 5, an instance of the diagram of figure 12 (without the object Version) is an implicit parameter of the sequents belonging to the partial evaluation system, to simplify the presentation of the sequents. Except for the object Version that is treated separately and appears in the sequents, the objects of the instance are never modified by the sequents.

4.2. Natural semantics rule for the partial evaluation

The inference rules for partial evaluation consist of sequents of the form

$$\mathsf{E,S,Cl,V} \overset{PE}{\vdash} \mathsf{I} \mapsto \mathsf{I}', \mathsf{S}', \mathsf{V}'$$

meaning that given the environment E, the state S, the inherited common blocks Cl and the set of specialized versions V, the specialization of the statement I yields a simplified statement I′, a new state S′ and a new set of versions V′. V′ is the union between V and versions that have been created during the simplification of I. The partial evaluation relation *PE* is the combination of the propagation (*propag*) and simplification (*simpl*) relations.

The dynamic semantic propagates all values of variables through statements. The propagation relation is close to the dynamic semantics relation except that it propagates only known values of variables (with respect to the initial values given by the user). The inference rules for propagation build the *propag* system. They consist of sequents of the form $\mathsf{E, S, Cl} \overset{propag}{\vdash} \mathsf{I : S}'$ meaning that given E, S and Cl, the execution of the statement I modifies the initial state S into the final state S′. The inference rules for simplification consist of sequents of the form $\mathsf{E, S, Cl, V} \overset{simpl}{\vdash} \mathsf{I} \hookrightarrow \mathsf{I}', \mathsf{V}'$ meaning that given E, S, Cl and V, the statement I simplifies into the statement I′ and the set of versions V becomes V′.

In the sequel of this section, we detail the rules for the propagation and the simplification of call statements. Then, we show how the rule for partial evaluation of call statements combines the propagation and the simplification rules.

4.2.1. Propagation of call statements. This section discusses how we have manually derived the propagation of a call statement from its dynamic semantics. This process is similar to the process described by Field, Ramalingam and Tip in (Field et al., 1995), where program slicing algorithms have been automatically derived from semantic specifications, using term rewriting systems.

Figure 13 specifies the propagation of a call statement to a procedure SP with a list LParam of actual parameters, given an environment E, a state S and inherited common blocks Cl. E, S and Cl are propagated through the statement `CALL` SP(LParam), and the result of the propagation is a new state S′.

As in the dynamic semantics, the propagation is composed of forward propagation followed by a backward propagation. Compared to the dynamic semantics, some parameters and variables have unknown values. Formal parameters of SP that have a known value are denoted by StaticFormal. As in figure 6, the map from the formal

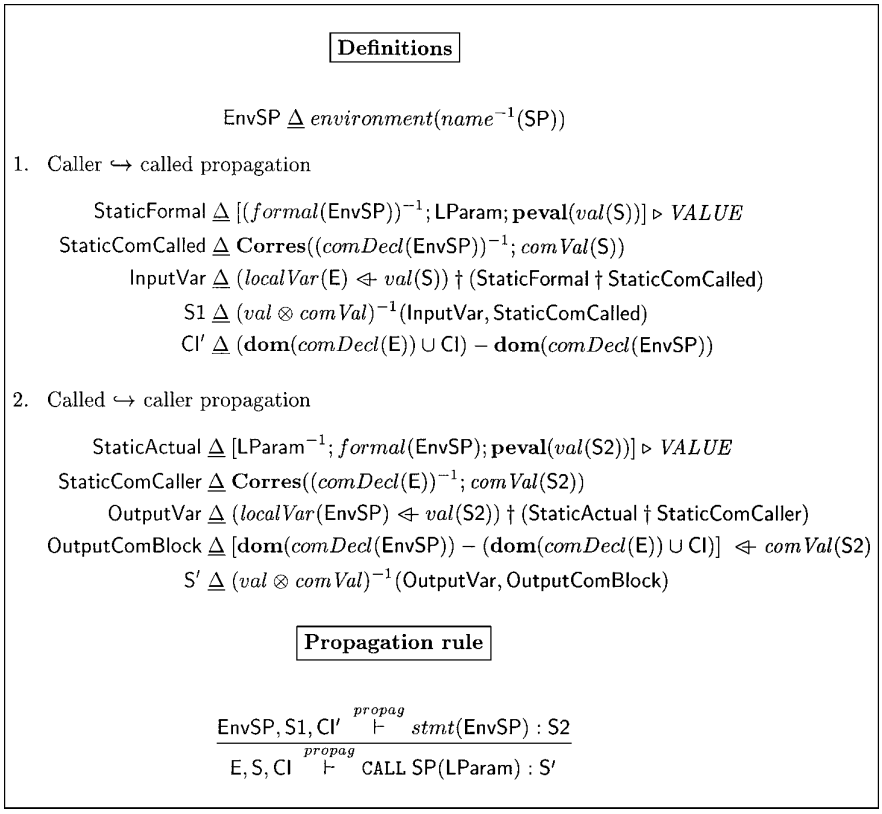


Figure 13. Propagation of call statements.

parameters of SP to the actual parameters of the current call statement is represented by $(formal(\mathsf{EnvSP}))^{-1}$; LParam. The **peval** function either yields the value of an expression (if it is known) or gives a residual expression (Blazy and Facon, 1994). Actual parameters are also partially evaluated and simplified (e.g. $x + 1 + y$ is simplified into the normalized expression $3 + x$ when the value of y is 2 and the value of x is unknown) but here evaluation yields expressions whose values may be unknown. As our partial evaluation propagates only equalities between variables and values, the resulting map is restricted to static formal parameters (i.e. formal parameters that have been totally evaluated).

In like manner, during the backward propagation process, StaticActual is computed in the following way:

1. LParam^{-1}; $formal(\mathsf{EnvSP})$ is a map of pairs (actual parameter, corresponding formal parameter).
2. Formal parameters are evaluated with respect to the known values at the end of the propagation in SP.
3. Dynamic actual parameters (they map to an expression whose value is unknown e.g. $z = a - 2$ with a unknown) are removed from the map. The resulting map is StaticActual.

Actually, the computations are the same as the dynamic semantic, except the computations of StaticFormal and StaticActual. The differences come from the evaluation of parameters. Blazy and Facon (1996) give examples of expressions propagated through called procedures.

4.2.2. Simplification of call statements. Figure 14 specifies the simplification of a call statement when the name of the called procedure is SP. The simplification rule:

- simplifies the statements of the called procedure (first premise of the rule),
- propagates data through these simplified statements (second premise),
- creates a new version of the called procedure (third premise).

In figure 14, V denotes the initial set of already specialized versions. As these versions are maintained (following the strategy introduced in Section 4.1), the implementation of the simplification rule has been optimized (see (Blazy and Facon, 1996) for details) to take into account the cases when:

- the called procedure has already been specialized with respect to the same static data,
- the most specialized version of the called procedure is not as specialized as wanted.

As for the dynamic semantics, the simplification process starts with the computation of a new state S1 and new inherited common blocks Cl′ (definitions of figure 14). These definitions are given in figure 13 and thus, they are not repeated in figure 14. As explained previously in the dynamic semantics, S1 represents the set of initial static variables of SP and Cl′ represents the inherited common blocks of SP.

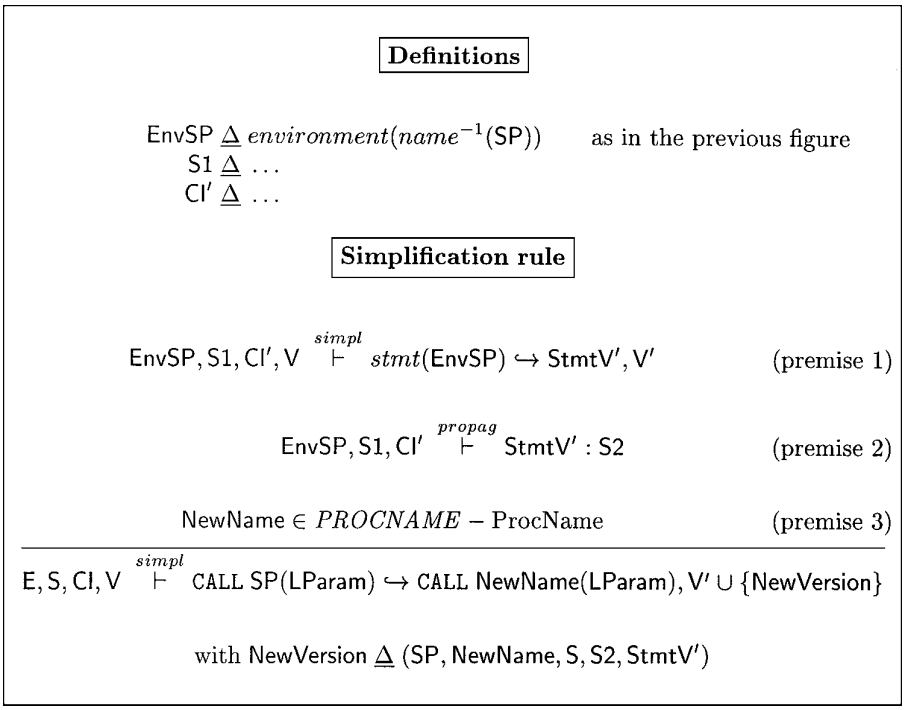


Figure 14. Simplification of call statements.

Then, given EnvSP, S1 and Cl′, the statements of the called procedure SP are simplified into StmtV′, yielding an updated set of specialized versions V′ (first premise). V′ is the union of V and the versions that have been created during the simplification of the statements of SP. Then, EnvSP, S1 and Cl′ are propagated through these simplified statements StmtV′, yielding a new state S2 for SP (second premise). A new name (NewName) is created for StmtV′ (third premise). This new name is a possible name that is not already a procedure name:

NewName ∈ *PROCNAME* − ProcName.

The call to SP is replaced by the call to NewName with the same parameters (no unfolding). The new specialized version NewVersion is also added among specialized versions of SP (conclusion of the rule). This new version is the quintuplet (*name of original procedure* = SP, *version name* = NewName, *input data* = S, *output data* = S2, *statements* = StmtV′).

4.2.3. Partial evaluation. Given an environment E, a state S, inherited common blocks Cl, and specialized versions V, the partial evaluation of a call statement combines the

$$\frac{\mathsf{E},\mathsf{S},\mathsf{CI} \overset{propag}{\vdash} \texttt{CALL}\ (\mathsf{LParam}) : \mathsf{S}' \qquad \mathsf{E},\mathsf{S},\mathsf{CI},\mathsf{V} \overset{simpl}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \hookrightarrow \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}),\mathsf{V}'}{\mathsf{E},\mathsf{S},\mathsf{CI},\mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}),\mathsf{S}',\mathsf{V}'}$$

Figure 15. Partial evaluation of call statements.

propagation of E, S and CI through this statement and the simplification of this statement. Figure 15 shows how the partial evaluation of a call statement yields a new call statement (with the same actual parameters), a new state and a new set of specialized versions.

4.3. Correctness of the partial evaluation

The natural semantics rules that specify partial evaluation define inductively a formal system (*PE*), that groups the propagation (*propag*) and the simplification (*simpl*) systems. Propagated data and simplified statements are built up by applying the natural semantics rules. To prove that the partial evaluation system is correct means proving that it is sound (each result of a residual program is correct with respect to the initial program) and complete (each correct result is computed by the residual program, too) with respect to the dynamic semantics of Fortran 90 given in Section 3. The derivation of the partial evaluation rules from the dynamic semantics rules has facilitated the proof of correctness. The dynamic semanatics rules are not proved: they are supposed to define *ex nihilo* the semantics of Fortran 90.

Figure 16 gives the general schema to prove the correctness of the partial evaluation. Given an initial procedure P, its environment E represents its syntax, and its inherited common blocks CI can be computed from its call graph. Initially, not a single specialized version has been created. Given some values S0 for known input data, partial evaluation of P yields a residual procedure P'. Let a state S' and a set of specialized versions V' be such that the sequent $\mathsf{E}, \mathsf{S0}, \mathsf{CI}, [\ ] \overset{PE}{\vdash} \mathsf{P} \mapsto \mathsf{P}', \mathsf{S}', \mathsf{V}'$ holds in the partial evaluation system (*PE*). In this sequent [ ] denotes the initial and empty list of specialized versions. Given this sequent, we want to prove that the partial evaluation is sound and complete with respect to

Given the sequent $\mathsf{E},\mathsf{S0},\mathsf{CI},[\,] \overset{PE}{\vdash} \mathsf{P} \mapsto \mathsf{P}',\mathsf{S}',\mathsf{V}'$
we want to show $\mathsf{E},\mathsf{S0}\cup\mathsf{S},\mathsf{CI} \overset{sem}{\vdash} \mathsf{P} : \mathsf{S}'' \Leftrightarrow \mathsf{E},\mathsf{S0}\cup\mathsf{S},\mathsf{CI} \overset{sem}{\vdash} \mathsf{P}' : \mathsf{S}''$
for all states S and S''

Figure 16. Correctness of the partial evaluation.

the dynamic semantics (*sem* system). This is expressed by the following property of partial evaluation, where the implication $\Leftarrow$ (resp. $\Rightarrow$) states soundness (resp. completeness).

$$\forall \mathsf{S}, \mathsf{S1} : (\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \mathsf{P} : \mathsf{S1} \Leftrightarrow \mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \mathsf{P}' : \mathsf{S1}).$$

S denotes the values of dynamic input variables (initially, their values are unknown). Such variables are the remaining input data once S0 has been given by the user. Thus, S0 ∪ S denotes the values of the whole input variables. To prove in the *sem* system this property on programs, we prove it for any Fortran 90 statement we simplify (i.e. for any rule of the partial evaluation system). Such a proof proceeds by structural induction on the Fortran 90 abstract syntax: the proof that the partial evaluation of a compound statement such as `IF` *c* `THEN` *i*1 `ELSE` *i*2 is correct is done under the induction hypothesis that states it is correct for the substatements *i1* and *i2*. The proof for some simple simplifications of statements has been given in Blazy and Facon (1995).

Our approach to prove the correctness of partial evaluation is close to the approach of Despeyroux (1986) to prove the correctness of translators: in that paper, dynamic semantics and translation are both given by formal systems and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules that are proved by induction on the length of the derivation of the translated terms. However, as we are concerned with only one language (Fortran 90), to simplify our proofs we do not deal with the validity of inference rules in the union of several formal systems (expressing the dynamic semantics of two langages and the translation from one language to the other). Furthermore, we do not need rule induction for all our proofs, but only structural induction and sometimes induction on derivations (to handle loops).

As an example, we give here the guidelines for proving by structural induction the soundness of the partial evaluation of a call statement `CALL` SP(LParam). The partial evaluation of this statement generates a call statement to a different procedure SP′, but SP′ is called with the same actual parameters as SP (i.e. LParam). Figure 17 gives the soundness rule to prove. This rule is proved under the induction hypothesis that the partial evaluation of the statements of SP is sound. The partial evaluation of this statement generates a call statement to a different procedure SP′, but SP′ is called with the same actual parameters as SP (see Section 4.2.3).

$$\frac{\mathsf{E}, \mathsf{S0}, \mathsf{CI}, \mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}), \mathsf{V}', \mathsf{S}' \qquad \mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}) : \mathsf{S}''}{\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI}, \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) : \mathsf{S}''}$$

Figure 17. Soundness rule for a call statement.

Given the two sequents of the premise of the rule given in figure 17, we have to prove that given the same input values $S0 \cup S$, the call statement of the source program yields the same output values S''. This is expressed by the conclusion of the rule. It will then prove the soundness of the partial evaluation of the call statement `CALL SP(LParam)` and of the specialized versions of SP. This is proved in two stages.

1. First, no data declaration is modified during the partial evaluation. Thus, both procedures SP and SP′ have the same formal parameters and the same data declarations. It follows that given input values $S0 \cup S$ the data declarations of SP yield the same values as the declarations of SP′.
2. Second, we assume that the induction hypothesis on the simplified statements of SP holds, and deduce that the new version and the statements of SP are sound as required.

5. The tool

We have implemented our partial evaluator on top of a kernel that has been generated by the Centaur system (INRIA, 1994). The Centaur system is a generic programming environment parametrized by the syntax and semantics of programming languages. When provided with the description of a particular programming language, Centaur produces a language-specific environment. The intermediate format for representing program text is the abstract syntax tree. We have merged two specific environments (one dedicated to Fortran 90 and another to a language that we have defined for expressing the scope of general constraints on variables) into an environment for partial evaluation. This environment consists of structured editors for constraints and Fortran 90 procedures (provided by Centaur), a partial evaluator, together with an uniform graphical interface. Figure 18 shows the architecture of our tool, its inputs and outputs. Given a file of constraints and an initial program represented by several

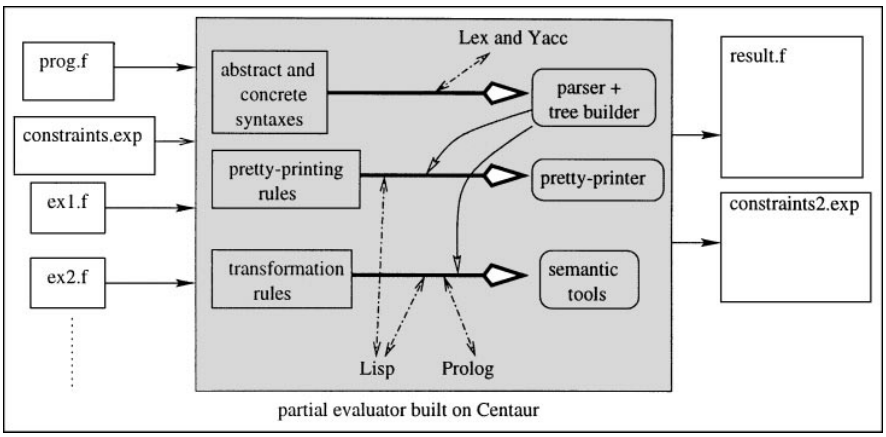


Figure 18. Architecture of the partial evaluator.

Fortran 90 files, the tool generates the program specialized with respect to these constraints and the corresponding final constraints.

The formal specifications have been implemented in a language provided by Centaur called Typol. Typol is an implementation of natural semantics. Typol programs are compiled into Prolog code. When executing these programs, Prolog is used as the engine of the deductive system. Set and relational operators as definitions have been written directly in Prolog in order to develop succinct and efficient Typol rules (Dubois and Sayarath, 1996). Thus, the Typol rules operate on the abstract syntax and they are close to the formal specification rules as shown in Blazy and Facon (1994). The partial evaluation process transforms an initial abstract syntax tree (representing the initial Fortran procedures and constraints) into a residual abstract syntax tree (representing the specialized code and the final constraints).

The abstract syntax used to describe Fortran 90 is general and close to the abstract syntax of any imperative language. For instance, to be more general, our specifications assume a dereferencing operator that does not exist in Fortran 90. The only peculiarities of Fortran 90 are the parameter passing (by reference only) and the use of common blocks instead of global variables. Except the corresponding specification rules, other rules are abstract enough to be the rules of any other imperative language (without recursion).

We have implemented a graphical interface to facilitate the exploration of Fortran 90 application programs (Vassallo, 1996). It has been written in Lisp, enhanced with structures for programming communication between graphical objects and processes. Different kinds of information are important for the user. At the beginning of the partial evaluation process, the user selects an initial application program (usually consisting of several Fortran files) and some constraints related to variables defined in any of the Fortran procedures. Then, the user runs the partial evaluator on his application program and constraints. Several partial evaluation processes can be run at once. Partial evaluation produces:

- the specialized application program (it has the same structure than the initial application program) and the links between this application program and the initial one,
- the specialized versions of called procedures and the links with the initial and final application programs,
- the final values of the variables whose initial values have been given by the user,
- the final values of other variables if they are known,
- information that is computed during the partial evaluation but that is not very important for most users (e.g. initial values of formal parameters before the partial evaluation of a called procedure).

The graphical interface organizes this information into different displays so that it is useful to the user. It allows the user:

- to identify specific nodes in abstract syntax trees (e.g. called procedures) as they are specifically colored,
- to display by default only the most important information,
- to trigger the display of other information when needed.

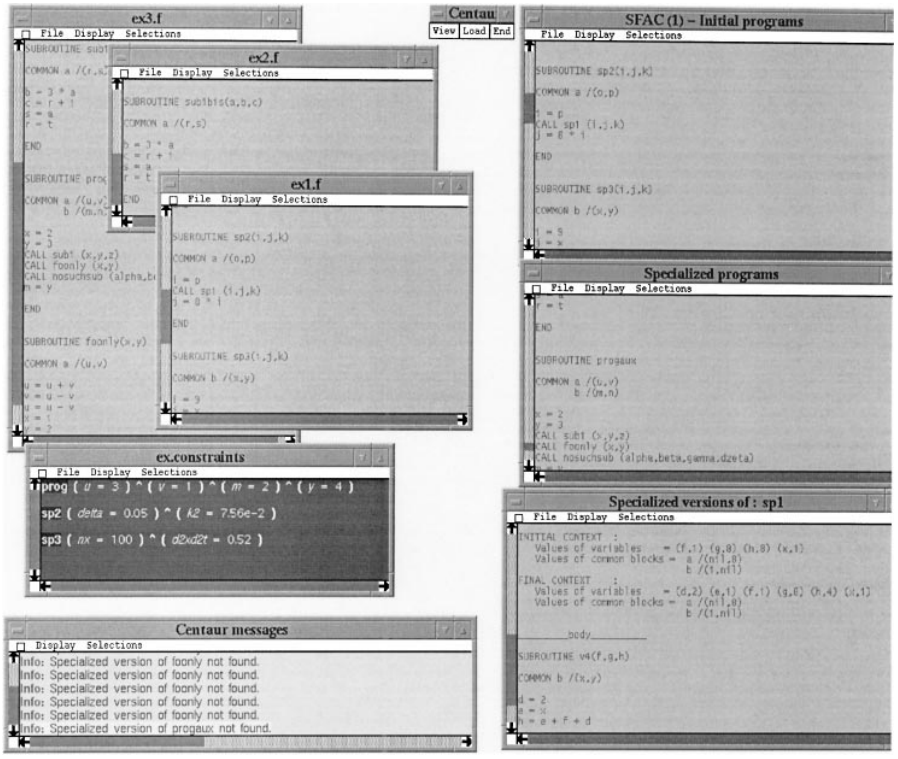


Figure 19. Partial evaluation of a Fortran 90 application (with reuse of specialized versions).

For instance, the specialized versions of a called procedure `P` are displayed only when the user clicks on a call statement to `P`. In the same way, the final values of propagated data are not displayed automatically.

Figure 19 presents the graphical interface. In this example:

- The three windows in the upper left corner show that the user has selected the files called `ex1.f`, `ex2.f` and `ex3.f`. They define the application program to specialize.
- The window below these three windows shows that the user has clicked in one of the three previous windows to choose the constraints related to these three windows. In figure 19 the constraints are written in the file called `ex.constraints`.
- The window in the lower left corner (called "Centaur messages") displays warning messages. In the example, the messages have been displayed when the user has asked for the display of a specialized version that does not exist. Error messages are also displayed, for instance when the user has forgotten to select constraints related to an initial program.
- The two windows in the upper right corner show that partial evaluation has been triggered (by a click in any of the three Fortran windows or in the constraints window). The first window resulting from the partial evaluation (called "Initial program" in figure 19)

displays all the procedures to specialize. The second window displays (below the previous one in the figure) all the specialized procedures.

- The lower right window displays the specialized versions of the procedure `sp1` (e.g. in the window called "Specialized versions of sp1" in figure 19). The display of this window has been triggered by a click on the call statement to `SP1` in any of the Fortran windows.

Figure 19 shows only one partial evaluator running on a constraints file and a Fortran application program made of three Fortran files. In figure 19, this partial evaluator is numbered SFAC(1) (this number is written in the title of the "Initial programs" window). But, several partial evaluations can also run in parallel. This is very useful when for instance the user needs to compare the results of the partial evaluation of an application program with several constraints files.

6. Conclusion

This paper has presented an approach to the understanding of application programs during their maintenance. The approach relies on partial evaluation, a technique that we have adapted to program understanding. The partial evaluation performs an interprocedural pointer analysis. We have formally specified our partial evaluation process and we have derived the propagation rules from the dynamic semantics rules, also expressed in natural semantics. In these specifications, inference rules in natural semantics show only how statements are simplified from data propagation and simplification of other statements. To clarify the presentation of the natural semantics rules, we have used set and relational operators to express outside the rules the computations on propagated data. Some of these data are not introduced directly in the dynamic semantics rules. They are accessed from other data, according to the object diagram that we have defined.

From the specifications, we have proven by induction the correctness of the partial evaluation with respect to a dynamic semantics of Fortran 90. This proof has been done by hand. We are currently investigating an automatic proof of the correctness of the partial evaluation.

A tool has been implemented from the specifications. A graphical interface has also been implemented to visualize program dependencies (mainly between variables and values and between reused versions of procedures). The tool has been tested at the EDF (the French national company that produces and distributes electricity), that provided us with scientific application programs (Blazy and Facon, 1994). The first results are very encouraging. We are planning more empirical work to validate these preliminary results: we intend to test other application programs using pointers extensively and to use metrics such as the metrics defined in Carini and Hind (1995) to evaluate our interprocedural constant propagation. Another current focus is in improving the analysis by propagating general constraints between variables instead of only equalities between variables and values. To this end, we could adapt the rules described in Bergstra et al. (1997).

Furthermore, partial evaluation is complementary to program slicing, another technique for extracting code when debugging a program. Program slicing aims at identifying the statements of the program which impact directly or indirectly on some variable values. We

believe that merging partial evaluation (a forward walk through the call graph) and program slicing (a blackward walk that can also be followed by a forward walk) would improve a lot the simplification of programs.

Acknowledgments

I would like to thank the anonymous reviewers for their detailed comments and suggestions, which have helped improve both the contents and presentation of this article.

In Memoriam

I would like to dedicate this paper to Professor Philippe Facon, who died tragically on September 1st, 1999. He was the leader of our research team and was one of the guiding forces of the project presented above.

Note

1. We could have used the other symbol for composition $\circ$ that is tantamount to ; since for any pair of binary relations r and p it is defined by $p \circ r = r; p$.

References

1. Abrial, J.R. 1996. *The B-Book Assigning Programs to Meanings*. Cambridge University Press, New York.
2. ACM. 1998. *Symposium on Partial Evaluation*, number 4 in ACM Computing Surveys.
3. Aho, A.V., Sethi, R., and Ullman, J.D. 1988. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.
4. Andersen, L.O., 1994. Program Analysis and Specialization for the C Programming Language. PhD Thesis, University of Copenhagen, DIKU report 94/19.
5. ANSI, 1992. New York. *Programming Language Fortran 90*, ANSI X3.198-1992 and ISO/IEC 1539–1991 (E).
6. Baier, R., Glück, R., and Zöchling, R. 1994. Partial evaluation of numerical programs in Fortran. In *Proceedings of the Partial Evaluation and Semantics Based Program Manipulation Workshop*, Melbourne, pp. 119–132. ACM SIGPLAN.
7. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. 1998. Clone detection using abstract syntax trees. In *Proc. of the Conf. on Soft. Maintenance*. IEEE.
8. Bergstra, J.A., Dinesh, T.B., Field, J., and Heering, J. 1997. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programing Languages and Systems*, 19(5):639–684.
9. Blazy, S. and Facon, P. 1994. SFAC, a tool for program comprehension by specialization. In *Proceedings of the Third Workshop on Program Comprehension*, Washington D.C., pp. 162–167. IEEE.
10. Blazy, S. and Facon, P. 1995. Formal specification and prototyping of a program specializer. In *Proceedings of the TAPSOFT Conference*, Lecture Notes in Computer Science, 915, pp. 666–680. Aarhus, Elsevier Science Publishers B.V. (North-Holland), Amsterdam.
11. Blazy, S. and Facon, P. 1996. An automatic interprocedural analysis for the understanding of scientific application programs. In Danvy et al. (1996), pp. 1–16.
12. Blazy, S. and Facon, P. 1998. Partial evaluation for program understanding. *ACM Computing Surveys—Symposium on Partial Evaluation*, 4(4).
13. Carini, R. and Hind, M. 1995. Flow-sensitive interprocedural constant propagation. In *Proc. of the Programming Languages Design and Implementation Conf.*, La Jolla, pp. 23–31. ACM SIGPLAN.

14. Chase, D.R., Wegman, M., and Zadeck, F.K. 1990. Analysis of pointers and structures. In *Proc. of the Programming Languages Design and Implementation Conf.*, White Plains, pp. 296–310. ACM SIGPLAN.
15. Danvy, O., Glück, R., and Thiemann, P. (Eds.), 1996. *International seminar on partial evaluation*, Dagstuhl castle, Elsevier Science Publishers B.V. (North-Holland), Amsterdam. Lecture Notes in Computer Science 1110.
16. Despeyroux, J. 1986. Proof of translation in natural semantics. In *Proceedings of the Symposium on Logic in Computer Science*, Cambridge, USA.
17. Dubois, N. and Sayarath, P. 1996. Aide à la compréhension et à la maintenance: Pointeurs pour la spécialisation de programmes. Master's Thesis. IIE-CNAM, in French.
18. Emami, M., Ghiya, R., and Hendren, L.J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the Programming Languages Design and Implementation Conf.*, pp. 242–256. ACM SIGPLAN.
19. Field, J., Ramalingam, G., and Tip, F. 1995. Parametric program slicing. In *Proc. of Principles of Programming Languages Conf.*, San Francisco, USA, pp. 379–392.
20. Hannan, J. 1993. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152.
21. Hasti, R. and Horwitz, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proc. of the Programming Languages Design and Implementation Conf.*, Montreal, pp. 97–105. ACM SIGPLAN.
22. INRIA, 1994. Centaur 1.2 documentation.
23. Jones, C.B. 1990. *Systematic Development Using VDM*. Prentice-Hall, Englewood Cliff.
24. Jones, N.D., Gomard, C.K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliff.
25. Kahn, G. 1987. Natural semantics. In *Proc. of the Symp. on Theoretical Aspects of Comp. Science*, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Lecture Notes in Computer Science 247, pp. 237–257.
26. Landi, W. and Ryder, B.G. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the Programming Languages Design and Implementation Conf.*, pp. 235–248. ACM SIGPLAN.
27. Liang, D. and Harrold, M.J. 1999. Efficient points-to analysis for whole-program analysis. In *Proc. of ESEC/FSE Joint Conf.*, Toulouse, France, Lecture Notes in Computer Science 1687, pp. 199–215. ACM SIGSOFT.
28. Marlet, R., Thibault, S., and Consel, C. 1997. Mapping software architectures to efficient implementation via partial evaluation. In *Proceedings of the Automated Software Engineering Conference*, pp. 183–192. IEEE.
29. Muchnick, S.S. 1997. *Advanced Compiler-Design and Implementation*. Morgan Kaufmann. Los Altos.
30. Nielson, H.R. and Nielson, F. 1992. *Semantics with Application—A Formal Introduction*. John Wiley and Sons, New York.
31. Sagiv, M., Reps, T., and Wilhelm, R. 1997. Solving shape-analysis problems in languages with destructive updating. In *Proc. of Principles of Programming Languages Conf.*
32. Sellink, M.P.A. and Verhoef, C. 2000. Scaffolding for software renovation. In *Proc. of the Conf. on Soft. Maintenance*, Zürich. IEEE.
33. Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proc. of Principles of Programming Languages Conf.*, pp. 32–41.
34. van den Brand, M.G.J., Klint, P., and Verhoef, C. 1996. Reverse engineering and system renovation: an annotated bibliography. Technical Report P9603, University of Amsterdam, Programming Research Group.
35. Vassallo, R. 1996. Ergonomie et évolution d'un outil de compréhension de programmes. Master's thesis. IIE-CNAM, in French.
36. Wilson, R.P. and Lam, M.S. 1995. Efficient context-sensitive pointer analysis for c programs. In *Proc. of the Programming Languages Design and Implementation Conf.*, pp. 1–12. ACM SIGPLAN.
37. Yank, H., Luker, P., and Chu, W. 1997. Code understanding through program transformation for reusable component identification. In *Fifth Workshop on Program Comprehension Proceedings*. IEEE.

Formal Verification of a C Compiler Front-End

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy

INRIA Rocquencourt, 78153 Le Chesnay, France
{Sandrine.Blazy, Zaynah.Dargaye, Xavier.Leroy}@inria.fr

Abstract. This paper presents the formal verification of a compiler front-end that translates a subset of the C language into the Cminor intermediate language. The semantics of the source and target languages as well as the translation between them have been written in the specification language of the Coq proof assistant. The proof of observational semantic equivalence between the source and generated code has been machine-checked using Coq. An executable compiler was obtained by automatic extraction of executable Caml code from the Coq specification of the translator, combined with a certified compiler back-end generating PowerPC assembly code from Cminor, described in previous work.

1 Introduction

Formal methods in general and program proof in particular are increasingly being applied to safety-critical software. These applications create a strong need for on-machine formalization and verification of programming language semantics and tools such as compilers, type-checkers and static analyzers. In particular, formal operational semantics are required to validate the logic of programs (e.g. axiomatic semantics) used to reason about programs. As for tools, the formal certification of compilers—that is, a proof that the generated executable code behaves as prescribed by the semantics of the source program—is needed to ensure that the guarantees obtained by formal verification of the source program carry over to the executable code.

For high-level programming languages such as Java and functional languages, there exists a considerable body of on-machine formalizations and verifications of operational semantics and programming tools such as compilers and bytecode verifiers. Despite being more popular for writing critical embedded software, lower-level languages such as C have attracted less interest: several formal semantics for various subsets of C have been published, but few have been carried on machine. (See section 5 for a review.)

The work presented in this paper is part of an ongoing project that aims at developing a realistic compiler for the C language and formally verifying that it preserves the semantics of the programs being compiled. A previous paper [8] describes the verification, using the Coq proof assistant, of the back-end of this compiler, which generates moderately optimized PowerPC assembly code from a low-level, imperative intermediate language called Cminor. The present paper reports on the development and proof of semantic preservation in Coq of a C

J. Misra, T. Nipkow, and E. Sekerinski (Eds.): FM 2006, LNCS 4085, pp. 460–475, 2006.

front-end for this compiler: a translator from Clight, a subset of the C language, to Cminor. To conduct the verification, a precise operational semantics of Clight was formalized in Coq. Clight features all C arithmetic types and operators, as well as arrays, pointers, pointers to functions, and all C control structures except `goto` and `switch`.

From a formal methods standpoint, this work is interesting in two respects. First, compilers are complex programs that perform sophisticated symbolic computations. Their formal verification is challenging, requiring difficult proofs by induction that are beyond the reach of many program provers. Second, proving the correctness of a compiler provides an indirect but original way to validate the semantics of the source language. It is relatively easy to formalize an operational semantics, but much harder to make sure that this semantics is correct and captures the intended meaning of programs. Typically, extensive testing and manual reviews of the semantics are needed. In our experience, proving the correctness of a translator to a simpler, lower-level language detects many small errors in the semantics of the source and target languages, and therefore generates additional confidence in both.

The remainder of this paper is organized as follows. Section 2 describes the Clight language and gives an overview of its operational semantics. Section 3 presents the translation from Clight to Cminor. Section 4 outlines the proof of correctness of this translation. Related work is discussed in section 5, followed by conclusions in section 6.

2 The Clight Language and Its Semantics

2.1 Abstract Syntax

The abstract syntax of Clight is given in figure 1. In the Coq formalization, this abstract syntax is presented as inductive data types, therefore achieving a deep embedding of Clight into Coq.

At the level of types, Clight features all the integral types of C, along with array, pointer and function types; `struct`, `union` and `typedef` types are currently omitted. The integral types fully specify the bit size of integers and floats, unlike the semi-specified C types `int`, `long`, etc.

Within expressions, all C operators are supported except those related to structs and unions. Expressions may have side-effects. All expressions and their sub-expressions are annotated by their static types. We write a^τ for the expression a carrying type τ. These types are necessary to determine the semantics of type-dependent operators such as the overloaded arithmetic operators. Similarly, combined arithmetic-assignment operators such as `+=` carry an additional type σ (as in $(a_1 \texttt{+=}^\sigma a_2)^\tau$) representing the result type of the arithmetic operation, which can differ from the type τ of the whole expression.

At the level of statements, all structured control statements of C (conditional, loops, `break`, `continue` and `return`) are supported, but not unstructured statements (`goto`, `switch`, `longjmp`). Two kinds of variables are allowed: global variables and local `auto` variables declared at the beginning of a function.

Types:

signedness	::= `Signed` | `Unsigned`	
intsize	::= `I8` | `I16` | `I32`	
floatsize	::= `F32` | `F64`	
τ	::= `Tint`$(intsize, signedness)$ | `Tfloat`$(floatsize)$	
	| `Tarray`(τ, n) | `Tpointer`(τ) | `Tvoid` | `Tfunction`(τ^*, τ)	

Expressions annotated with types:

a	::= b^τ	

Unannotated expressions:

b	::= id	variable identifier
	| n | f	integer or float constant
	| `sizeof`(τ)	size of a type
	| $op_u\ a$	unary arithmetic operation
	| $a_1\ op_b\ a_2$ | $a_1\ op_r\ a_2$	binary arithmetic operation
	| `*`a	dereferencing
	| $a_1[a_2]$	array indexing
	| `&`a	address of
	| `++`a | `--`a | a`++` | a`--`	pre/post increment/decrement
	| $(\tau)a$	cast
	| $a_1 = a_2$	assignment
	| $a_1\ op_b =^\tau\ a_2$	arithmetic with assignment
	| a_1 `&&` a_2 | a_1 `||` a_2	sequential boolean operations
	| a_1, a_2	sequence of expressions
	| $a(a^*)$	function call
	| a_1 `?` a_2 `:` a_3	conditional expression
op_b	::= `+` | `-` | `*` | `/` | `%`	arithmetic operators
	| `<<` | `>>` | `&` | `|` | `^`	bitwise operators
op_r	::= `<` | `<=` | `>` | `>=` | `==` | `!=`	relational operators
op_u	::= `-` | `~` | `!`	unary operators

Statements:

s	::= `skip`	empty statement
	| a;	expression evaluation
	| $s_1; s_2$	sequence
	| `if`(a) s_1 `else` s_2	conditional
	| `while`(a) s	"while" loop
	| `do` s `while`(a)	"do" loop
	| `for`$(a_1^?, a_2^?, a_3^?)$ s	"for" loop
	| `break`	exit from the current loop
	| `continue`	next iteration of the current loop
	| `return` $a^?$	return from current function

Functions:

fn	::= $(\ldots id_i : \tau_i \ldots) : \tau$	declaration of type and parameters
	{ $\ldots \tau_j\ id_j; \ldots$	declaration of local variables
	s }	function body

Fig. 1. Abstract syntax of Clight. a^* denotes 0, 1 or several occurrences of syntactic category a. $a^?$ denotes an optional occurrence of category a.

Block-scoped local variables and `static` variables are omitted, but can be emulated by pulling their declarations to function scope or global scope, respectively. Consequently, there is no block statement in Clight.

A Clight program consists of a list of function definitions, a list of global variable declarations, and an identifier naming the entry point of the program (the `main` function in C).

2.2 Dynamic Semantics

The dynamic semantics of Clight is specified using natural semantics, also known as big-step operational semantics. While the semantics of C is not deterministic (the evaluation order for expressions is not completely specified and compilers are free to choose between several orders), the semantics of Clight is completely deterministic and imposes a left-to-right evaluation order, consistent with the order implemented by our compiler. This choice simplifies greatly the semantics compared with, for instance, Norrish's semantics for C [10], which captures the non-determinism allowed by the ISO C specification. Our semantics can therefore be viewed as a refinement of (a subset of) the ISO C semantics, or of that of Norrish.

The semantics is defined by 7 judgements (relations):

$$
\begin{array}{rl}
G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M' & \text{(expressions in l-value position)} \\
G, E \vdash a, M \Rightarrow v, M' & \text{(expressions in r-value position)} \\
G, E \vdash a^?, M \Rightarrow v, M' & \text{(optional expressions)} \\
G, E \vdash a^*, M \Rightarrow v^*, M' & \text{(list of expressions)} \\
G, E \vdash s, M \Rightarrow out, M' & \text{(statements)} \\
G \vdash f(v^*), M \Rightarrow v, M' & \text{(function invocations)} \\
\vdash p \Rightarrow v & \text{(programs)}
\end{array}
$$

Each judgement relates a syntactic element (expression, statement, etc) and an initial memory state to the result of executing this syntactic element, as well as the final memory state at the end of execution. The various kinds of results, as well as the evaluation environments, are defined in figure 2.

For instance, executing an expression a in l-value position results in a memory location loc (a memory block reference and an offset within that block), while executing an expression a in r-value position results in the value v of the expression. Values range over 32-bit integers, 64-bit floats, memory locations (pointers), and an undefined value that represents for instance the value of uninitialized variables. The result associated with the execution of a statement s is an "outcome" out indicating how the execution terminated: either normally by running to completion or prematurely via a `break`, `continue` or `return` statement. The invocation of a function f yields its return value v, and so does the execution of a program p.

Two evaluation environments, defined in figure 2, appear as parameters to the judgements. The local environment E maps local variables to references of memory blocks containing the values of these variables. These blocks are allocated

Values:

loc	$::= (b, n)$	location (byte offset n in block referenced by b)
v	$::= \texttt{Vint}(n)$	integer value
	$\mid \texttt{Vfloat}(f)$	floating-point value
	$\mid \texttt{Vptr}(loc)$	pointer value
	$\mid \texttt{Vundef}$	undefined value

Statement outcomes:

out	$::= \texttt{Out\_normal}$	go to the next statement
	$\mid \texttt{Out\_continue}$	go to the next iteration of the current loop
	$\mid \texttt{Out\_break}$	exit from the current loop
	$\mid \texttt{Out\_return}$	function exit
	$\mid \texttt{Out\_return}(v, \tau)$	function exit, returning the value v of type τ

Global environments:

G	$::= (id \mapsto b)$	map from global variables to block references
	$\times (b \mapsto fn)$	and map from references to function definitions

Local environments:

E	$::= id \mapsto b$	map from local variables to block references

Fig. 2. Values, outcomes, and evaluation environments

at function entry and freed at function return. The global environment G maps global variables and function names to memory references. It also maps some references (those corresponding to function pointers) to function definitions.

In the Coq specification, the 7 judgements of the dynamic semantics are encoded as mutually-inductive predicates. Each defining case of each predicate corresponds exactly to an inference rule in the conventional, on-paper presentation of natural semantics. We have one inference rule for each kind of expression and statement described in figure 1. We do not list all the inference rules by lack of space, but show some representative examples in figure 3.

The first two rules of figure 3 illustrate the evaluation of an expression in l-value position. A variable x evaluates to the location $(E(x), 0)$. If an expression a evaluates to a pointer value $\texttt{Vptr}(loc)$, then the location of the dereferencing expression $(*a)^\tau$ is loc.

Rule 3 evaluates an application of a binary operator op to expressions a_1 and a_2. Both sub-expressions are evaluated in sequence, and their values are combined with the `eval_binary_operation` function, which takes as additional arguments the types τ_1 and τ_2 of the arguments, in order to resolve overloaded and type-dependent operators. This is a partial function: it can be undefined if the types and the shapes of argument values are incompatible (e.g. a floating-point addition of two pointer values). In the Coq specification, `eval_binary_operation` is a total function returning optional values: either `None` in case of failure, or $\texttt{Some}(v)$, abbreviated as $\lfloor v \rfloor$, in case of success.

Rule 4 rule shows the evaluation of an l-value expression in a r-value context. The expression is evaluated to its location loc, with final memory state M'. The value at location loc in M' is fetched using the `loadval` function (see section 2.3) and returned.

Expressions in l-value position:

$$\frac{E(x) = b}{G, E \vdash x^\tau, M \stackrel{l}{\Rightarrow} (b, 0), M} \quad (1) \qquad \frac{G, E \vdash a, M \Rightarrow \mathtt{Vptr}(loc), M'}{G, E \vdash (*a)^\tau, M \stackrel{l}{\Rightarrow} loc, M'} \quad (2)$$

Expressions in r-value position:

$$\frac{G, E \vdash a_1^{\tau_1}, M \Rightarrow v_1, M_1 \quad G, E \vdash a_2^{\tau_2}, M_1 \Rightarrow v_2, M_2 \quad \mathtt{eval\_binary\_operation}(op, v_1, \tau_1, v_2, \tau_2) = \lfloor v \rfloor}{G, E \vdash (a_1^{\tau_1}\, op\, a_2^{\tau_2})^\tau, M \Rightarrow v, M_2} \quad (3)$$

$$\frac{G, E \vdash a^\tau, M \stackrel{l}{\Rightarrow} loc, M' \quad \mathtt{loadval}(\tau, M', loc) = \lfloor v \rfloor}{G, E \vdash a^\tau, M \Rightarrow v, M'} \quad (4)$$

$$\frac{G, E \vdash a^\tau, M \stackrel{l}{\Rightarrow} loc, M_1 \quad G, E \vdash b^\sigma, M_1 \Rightarrow v_1, M_2 \quad \mathtt{cast}(v_1, \sigma, \tau) = \lfloor v \rfloor \quad \mathtt{storeval}(\tau, M_2, loc, v) = \lfloor M_3 \rfloor}{G, E \vdash (a^\tau \ \mathtt{=}\ b^\sigma)^\tau, M \Rightarrow v, M_3} \quad (5)$$

Statements:

$$G, E \vdash \mathtt{break}, M \Rightarrow \mathtt{Out\_break}, M \quad (6)$$

$$\frac{G, E \vdash s_1, M \Rightarrow \mathtt{Out\_normal}, M_1 \quad G, E \vdash s_2, M_1 \Rightarrow out, M_2}{G, E \vdash (s_1; s_2), M \Rightarrow out, M_2} \quad (7)$$

$$\frac{G, E \vdash s_1, M \Rightarrow out, M' \quad out \neq \mathtt{Out\_normal}}{G, E \vdash (s_1; s_2), M \Rightarrow out, M'} \quad (8)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M' \quad \mathtt{is\_false}(v)}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow \mathtt{Out\_normal}, M'} \quad (9)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M_1 \quad \mathtt{is\_true}(v) \quad G, E \vdash s, M_1 \Rightarrow \mathtt{Out\_break}, M_2}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow \mathtt{Out\_normal}, M_2} \quad (10)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M_1 \quad \mathtt{is\_true}(v) \quad G, E \vdash s, M_1 \Rightarrow out, M_2 \quad out \in \{\mathtt{Out\_normal}, \mathtt{Out\_continue}\} \quad G, E \vdash (\mathtt{while}(a)\ s), M_2 \Rightarrow out', M_3}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow out', M_3} \quad (11)$$

Fig. 3. Selected rules of the dynamic semantics of Clight

Rule 5 evaluates an assignment expression. An assignment expression a^τ `=` b^σ evaluates the l-value a to a location loc, then the r-value b to a value v_1. This value is cast from its natural type σ to the expected type τ using the partial function `cast`. This function performs appropriate conversions, truncations and sign-extensions over integers and floats, and may fail for undefined casts. The result v of the cast is then stored in memory at location loc, resulting in the

final memory state M_3, and returned as the value of the assignment expression.

The bottom group of rules in figure 3 are examples of statement executions. The execution of a break statement yields an `Out_break` outcome (rule 6). The execution of a sequence of two statements starts with the execution of the first statement, yielding an outcome that determines if the second statement must be executed or not (rules 7 and 8). Finally, rules 9–11 describe the execution of a `while` loop. Once the condition of a while loop is evaluated to a value v, the execution of the loop terminates normally if v is false. If v is true, the loop body is executed, yielding an outcome *out*. If *out* is `Out_break`, the loop terminates normally. If *out* is `Out_normal` or `Out_continue`, the whole loop is reexecuted in the memory state modified by the execution of the body.

2.3 The Memory Model of the Semantics

The memory model used in the dynamic semantics is described in [1]. It is a compromise between a low-level view of memory as an array of bytes and a high-level view as a mapping from abstract references to contents. In our model, the memory is arranged in independent blocks, identified by block references b. A memory state M maps references b to block contents, which are themselves mappings from byte offsets to values. Each block has a low bound $L(M, b)$ and a high bound $H(M, b)$, determined at allocation time and representing the interval of valid byte offsets within this block. This memory model guarantees separation properties between two distinct blocks, yet enables pointer arithmetic within a given block, as prescribed by the ISO C specification. The same memory model is common to the semantics of all intermediate languages of our certified compiler.

The memory model provides 4 basic operations:

$\texttt{alloc}(M, lo, hi) = (M', b)$
: Allocate a fresh block of bounds $[lo, hi)$. Returns extended memory M' and reference b to fresh block.

$\texttt{free}(M, b) = M'$
: Free (invalidate) the block b.

$\texttt{load}(\kappa, M, b, n) = \lfloor v \rfloor$
: Read one or several consecutive bytes (as determined by κ) at block b, offset n in memory state M. If successful return the contents of these bytes as value v.

$\texttt{store}(\kappa, M, b, n, v) = \lfloor M' \rfloor$
: Store the value v into one or several consecutive bytes (as determined by κ) at offset n in block b of memory state M. If successful, return an updated memory state M'.

The memory chunks κ appearing in `load` and `store` operations describe concisely the size, type and signedness of the memory quantities involved:

$\kappa ::=$ `Mint8signed` | `Mint8unsigned`	
| `Mint16signed` | `Mint16unsigned`	small integers
| `Mint32`	integers and pointers
| `Mfloat32` | `Mfloat64`	floats

In the semantics of C, those quantities are determined by the C types of the datum being addressed. The following $\mathcal{A}$ ("access mode") function mediates between C types and the corresponding memory chunks:

$$\begin{aligned}
\mathcal{A}(\texttt{Tint}(\texttt{I8}, \texttt{Signed})) &= \texttt{By_value}(\texttt{Mint8signed}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I8}, \texttt{Unsigned})) &= \texttt{By_value}(\texttt{Mint8unsigned}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I16}, \texttt{Signed})) &= \texttt{By_value}(\texttt{Mint16signed}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I16}, \texttt{Unsigned})) &= \texttt{By_value}(\texttt{Mint16unsigned}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I32}, _)) = \mathcal{A}(\texttt{Tpointer}(_)) &= \texttt{By_value}(\texttt{Mint32}) \\
\mathcal{A}(\texttt{Tarray}(_, _)) = \mathcal{A}(\texttt{Tfunction}(_, _)) &= \texttt{By_reference} \\
\mathcal{A}(\texttt{Tvoid}) &= \texttt{By_nothing}
\end{aligned}$$

Integer, float and pointer types involve an actual memory `load` when accessed, as captured by the `By_value` cases. However, accesses to arrays and functions return the location of the array or function, without any `load`; this is indicated by the `By_reference` access mode. Finally, expressions of type `void` cannot be accessed at all. This is reflected in the definitions of the `loadval` and `storeval` functions used in the dynamic semantics:

$$\begin{aligned}
\texttt{loadval}(\tau, M, (b, n)) &= \texttt{load}(\kappa, M, b, n) && \text{if } \mathcal{A}(\tau) = \texttt{By_value}(\kappa) \\
\texttt{loadval}(\tau, M, (b, n)) &= \lfloor b, n \rfloor && \text{if } \mathcal{A}(\tau) = \texttt{By_reference} \\
\texttt{loadval}(\tau, M, (b, n)) &= \texttt{None} && \text{if } \mathcal{A}(\tau) = \texttt{By_nothing} \\
\texttt{storeval}(\tau, M, (b, n), v) &= \texttt{store}(\kappa, M, b, n, v) && \text{if } \mathcal{A}(\tau) = \texttt{By_value}(\kappa) \\
\texttt{storeval}(\tau, M, (b, n), v) &= \texttt{None} && \text{otherwise}
\end{aligned}$$

2.4 Static Semantics (Typing Rules)

We have also formalized in Coq typing rules and a type checking algorithm for Clight. The algorithm is presented as a function from abstract syntax trees without type annotations to the abstract syntax trees with type annotations over expressions given in figure 1. We omit the typing rules by lack of space. Note that the dynamic semantics are defined for arbitrarily annotated expressions, not just well-typed expressions; however, the semantics can get stuck or produce results that disagree with ISO C when given an incorrectly-annotated expression. The translation scheme presented in section 3 demands well-typed programs and may fail to preserve semantics otherwise.

3 Translation from Clight to Cminor

3.1 Overview of Cminor

The Cminor language is the target language of our front-end compiler for C and the input language for our certified back-end. We now give a short overview of Cminor; see [8] for a more detailed description, and [7] for a complete formal specification.

Cminor is a low-level imperative language, structured like our subset of C into expressions, statements, and functions. We summarize the main differences with Clight. First, arithmetic operators are not overloaded and their behavior is independent of the static types of their operands. Distinct operators are provided for integer arithmetic and floating-point arithmetic. Conversions between integers and floats are explicit. Arithmetic is always performed over 32-bit integers and 64-bit floats; explicit truncation and sign-extension operators are provided to implement smaller integral types. Finally, the combined arithmetic-with-assignment operators of C (`+=`, `++`, etc) are not provided. For instance, the C expression `i += f` where `i` is of type `int` and `f` of type `double` is expressed as `i = intoffloat(floatofint(i) +`$_f$ `f)`.

Address computations are explicit, as well as individual load and store operations. For instance, the C expression `a[i]` where `a` is a pointer to `int` is expressed as `load(int32, a +`$_i$ `i *`$_i$ `4)`, making explicit the memory chunk being addressed (`int32`) as well as the computation of the address.

At the level of statements, Cminor has only 4 control structures: if-then-else conditionals, infinite loops, `block-exit`, and early return. The `exit` n statement terminates the $(n + 1)$ enclosing `block` statements. These structures are lower-level than those of C, but suffice to implement all reducible flow graphs.

Within Cminor functions, local variables can only hold scalar values (integers, pointers, floats) and they do not reside in memory, making it impossible to take a pointer to a local variable like the C operator `&` does. Instead, each Cminor function declares the size of a stack-allocated block, allocated in memory at function entry and automatically freed at function return. The expression `addrstack`(n) returns a pointer within that block at constant offset n. The Cminor producer can use this block to store local arrays as well as local scalar variables whose addresses need to be taken.[1]

The semantics of Cminor is defined in big-step operational style and resembles that of Clight. The following evaluation judgements are defined in [7]:

$$\begin{aligned} G, sp, L \vdash a, E, M &\to v, E', M' && \text{(expressions)}\\ G, sp, L \vdash a^*, E, M &\to v^*, E', M' && \text{(expression lists)}\\ G, sp \vdash s, E, M &\to out, E', M' && \text{(statements)}\\ G \vdash fn(v^*), M &\to v, M' && \text{(function calls)}\\ \vdash prog &\to v && \text{(whole programs)} \end{aligned}$$

The main difference with the semantics of Clight is that the local evaluation environment E maps local variables to their values, instead of their memory addresses; consequently, E is modified during evaluation of expressions and statements. Additional parameters are sp, the reference to the stack block for the current function, and L, the environment giving values to variables `let`-bound within expressions.

[1] While suboptimal in terms of performance of generated code, this systematic stack allocation of local variables whose addresses are taken is common practice for moderately optimizing C compilers such as `gcc` versions 2 and 3.

3.2 Overview of the Translation

The translation from our subset of Clight to Cminor performs three basic tasks:

- Resolution of operator overloading and explication of all type-dependent behaviors. Based on the types that annotate Clight expressions, the appropriate flavors (integer or float) of arithmetic operators are chosen; conversions between ints and floats, truncations and sign-extensions are introduced to reflect casts, both explicit in the source and implicit in the semantics of Clight; address computations are generated based on the types of array elements and pointer targets; and appropriate memory chunks are selected for every memory access.
- Translation of `while`, `do...while` and `for` loops into infinite loops with blocks and early exits. The statements `break` and `continue` are translated as appropriate `exit` constructs, as shown in figure 4.
- Placement of Clight variables, either as Cminor local variables (for local scalar variables whose address is never taken), sub-areas of the Cminor stack block for the current function (for local non-scalar variables or local scalar variables whose address is taken), or globally allocated memory areas (for global variables).[2]

The translation is specified as Coq functions from Clight abstract syntax to Cminor abstract syntax, defined by structural recursion. From these Coq functions, executable Caml code can be mechanically generated using the Coq extraction facility, making the specification directly executable. Several translation functions are defined: $\mathcal{L}$ and $\mathcal{R}$ for expressions in l-value and r-value position, respectively; $\mathcal{S}$ for statements; and $\mathcal{F}$ for functions. Some representative cases of the definitions of these functions are shown in figure 4, giving the general flavor of the translation.

The translation can fail when given invalid Clight source code, e.g. containing an assignment between arrays. To enable error reporting, the translation functions return option types: either `None` denoting an error, or $\lfloor x \rfloor$ denoting successful translation with result x. Systematic propagation of errors is achieved using a monadic programming style (the `bind` combinator of the error monad), as customary in purely functional programming. This monadic "plumbing" is omitted in figure 4 for simplicity.

Most translation functions are parameterized by a translation environment γ reflecting the placement of Clight variables. It maps every variable x to either `Local` (denoting the Cminor local variable named x), `Stack`(δ) (denoting a subarea of the Cminor stack block at offset δ), or `Global` (denoting the address of the Cminor global symbol named x). This environment is constructed at the beginning of the translation of a Clight function. The function body is

[2] It would be semantically correct to stack-allocate all local variables, like the C0 verified compiler does [6,12]. However, keeping scalar local variables in Cminor local variables as much as possible enables the back-end to generate much more efficient machine code.

Casts ($\mathcal{C}^{\sigma}_{\tau}(e)$ casts e from type τ to type σ):

$$\mathcal{C}^{\sigma}_{\tau}(e) = \mathcal{C}_2(\mathcal{C}_1(e,\tau,\sigma),\sigma)$$

$$\mathcal{C}_1(e,\tau,\sigma) = \begin{cases} \texttt{floatofint}(e), & \text{if } \tau = \texttt{Tint}(\_,\texttt{Signed}) \text{ and } \sigma = \texttt{Tfloat}(\_); \\ \texttt{floatofintu}(e), & \text{if } \tau = \texttt{Tint}(\_,\texttt{Unsigned}) \text{ and } \sigma = \texttt{Tfloat}(\_); \\ \texttt{intoffloat}(e), & \text{if } \tau = \texttt{Tfloat}(\_) \text{ and } \sigma = \texttt{Tint}(\_,\_); \\ e & \text{otherwise} \end{cases}$$

$$\mathcal{C}_2(e,\sigma) = \begin{cases} \texttt{cast8signed}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I8},\texttt{Signed}); \\ \texttt{cast8unsigned}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I8},\texttt{Unsigned}); \\ \texttt{cast16signed}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I16},\texttt{Signed}); \\ \texttt{cast16unsigned}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I16},\texttt{Unsigned}); \\ \texttt{singleoffloat}(e), & \text{if } \sigma = \texttt{Tfloat}(\texttt{F32}); \\ e & \text{otherwise} \end{cases}$$

Expressions in l-value position:

$$\mathcal{L}_\gamma(x) = \texttt{addrstack}(\delta) \quad \text{if } \gamma(x) = \texttt{Stack}(\delta)$$
$$\mathcal{L}_\gamma(x) = \texttt{addrglobal}(x) \quad \text{if } \gamma(x) = \texttt{Global}$$
$$\mathcal{L}_\gamma(\texttt{*}e) = \mathcal{R}_\gamma(e)$$
$$\mathcal{L}_\gamma(e_1[e_2]) = \mathcal{R}_\gamma(e_1 \texttt{ + } e_2)$$

Expressions in r-value position:

$$\mathcal{R}_\gamma(x) = x \quad \text{if } \gamma(x) = \texttt{Local}$$
$$\mathcal{R}_\gamma(e^\tau) = \texttt{load}(\kappa, \mathcal{L}_\gamma(e^\tau)) \quad \text{if } \mathcal{L}_\gamma(e) \text{ is defined and } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa)$$
$$\mathcal{R}_\gamma(e^\tau) = \mathcal{L}_\gamma(e^\tau) \quad \text{if } \mathcal{L}_\gamma(e) \text{ is defined and } \mathcal{A}(\tau) = \texttt{By\_reference}$$
$$\mathcal{R}_\gamma(x^\tau \texttt{ = } e^\sigma) = x \texttt{ = } \mathcal{C}^\tau_\sigma(\mathcal{R}(e^\sigma)) \quad \text{if } \gamma(x) = \texttt{Local}$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ = } e_2^\sigma) = \texttt{store}(\kappa, \mathcal{L}_\gamma(e_1^\tau), \mathcal{C}^\tau_\sigma(\mathcal{R}(e_2^\sigma))) \quad \text{if } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa)$$
$$\mathcal{R}_\gamma(\texttt{\&}e) = \mathcal{L}_\gamma(e)$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ + } e_2^\sigma) = \mathcal{R}_\gamma(e_1^\tau) \texttt{ +}_i \mathcal{R}_\gamma(e_2^\sigma) \quad \text{if } \tau \text{ and } \sigma \text{ are integer types}$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ + } e_2^\sigma) = \mathcal{C}^{\texttt{double}}_\tau(\mathcal{R}_\gamma(e_1^\tau)) \texttt{ +}_f \mathcal{C}^{\texttt{double}}_\sigma(\mathcal{R}_\gamma(e_2^\sigma)) \quad \text{if } \tau \text{ or } \sigma \text{ are float types}$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ + } e_2^\sigma) = \mathcal{R}_\gamma(e_1^\tau) \texttt{ +}_i \mathcal{R}_\gamma(e_2^\sigma) \texttt{ *}_i \texttt{ sizeof}(\rho) \quad \text{if } \tau \text{ is a pointer or array of } \rho$$

Statements:

$$\mathcal{S}_\gamma(\texttt{while}(e)\ s) = \texttt{block\{ loop\{ if } (\texttt{!}\mathcal{R}_\gamma(e)) \texttt{ exit 0; block\{ } \mathcal{S}_\gamma(s) \texttt{ \}\}\}}$$
$$\mathcal{S}_\gamma(\texttt{do } s \texttt{ while}(e)) = \texttt{block\{ loop\{ block\{ } \mathcal{S}_\gamma(s) \texttt{ \}; if } (\texttt{!}\mathcal{R}_\gamma(e)) \texttt{ exit 0 \}\}}$$
$$\mathcal{S}_\gamma(\texttt{for}(e_1;e_2;e_3)\ s) = \mathcal{R}_\gamma(e_1); \texttt{ block\{ loop\{ if } (\texttt{!}\mathcal{R}_\gamma(e_2)) \texttt{ exit 0; block\{ } \mathcal{S}_\gamma(s) \texttt{ \}; } \mathcal{R}_\gamma(e_3) \texttt{ \}\}}$$
$$\mathcal{S}_\gamma(\texttt{break}) = \texttt{exit 1}$$
$$\mathcal{S}_\gamma(\texttt{continue}) = \texttt{exit 0}$$

Fig. 4. Selected translation rules

scanned for occurrences of `&x` (taking the address of a variable). Local variables that are not scalar or whose address is taken are assigned $\texttt{Stack}(\delta)$ locations, with δ chosen so that distinct variables map to non-overlapping areas of the stack block. Other local variables are set to `Local`, and global variables to `Global`.

4 Proof of Correctness of the Translation

4.1 Relating Memory States

To prove the correctness of the translation, the major difficulty is to relate the memory states occurring during the execution of the Clight source code and that of the generated Cminor code. The semantics of Clight allocates a distinct block for every local variable at function entry. Some of those blocks (those for scalar variables whose address is not taken) have no correspondence in the Cminor memory state; others become sub-block of the Cminor stack block for the function.

To account for these differences in allocation patterns between the source and target code, we introduce the notion of *memory injections*. A memory injection α is a function from Clight block references b to either `None`, meaning that this block has no counterpart in the Cminor memory state, or $\lfloor b', \delta \rfloor$, meaning that the block b of the Clight memory state corresponds to a sub-block of block b' at offset δ in the Cminor memory state.

A memory injection α defines a relation between Clight values v and Cminor values v', written $\alpha \vdash v \approx v'$ and defined as follows:

$$\alpha \vdash \texttt{Vint}(n) \approx \texttt{Vint}(n) \qquad \alpha \vdash \texttt{Vfloat}(n) \approx \texttt{Vfloat}(n) \qquad \alpha \vdash \texttt{Vundef} \approx v$$

$$\frac{\alpha(b) = \lfloor b', \delta \rfloor \qquad i' = i + \delta \pmod{2^{32}}}{\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')}$$

This relation captures the relocation of pointer values implied by α. It also enables `Vundef` Clight values to become more defined Cminor values during the translation, in keeping with the general idea that compilation can particularize some undefined behaviors.

The memory injection α also defines a relation between Clight and Cminor memory states, written $\alpha \vdash M \approx M'$, consisting of the conjunction of the following conditions:

- Matching of block contents: if $\alpha(b) = \lfloor b', \delta \rfloor$ and $L(M, b) \leq i < H(M, b)$, then $L(M', b') \leq i + \delta < H(M', b')$ and $\alpha \vdash v \approx v'$ where v is the contents of block b at offset i in M and v' the contents of b' at offset i' in M'.
- No overlap: if $\alpha(b_1) = \lfloor b'_1, \delta_1 \rfloor$ and $\alpha(b_2) = \lfloor b'_2, \delta_2 \rfloor$ and $b_1 \neq b_2$, then either $b'_1 \neq b'_2$, or the intervals $[L(M, b_1) + \delta_1, H(M, b_1) + \delta_1)$ and $[L(M, b_2) + \delta_2, H(M, b_2) + \delta_2)$ are disjoint.
- Fresh blocks: $\alpha(b) = \texttt{None}$ for all blocks b not yet allocated in M.

The memory injection relations have nice commutation properties with respect to the basic operations of the memory model. For instance:

- Commutation of loads: if $\alpha \vdash M \approx M'$ and $\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')$ and $\texttt{load}(\kappa, M, b, i) = \lfloor v \rfloor$, there exists v' such that $\texttt{load}(\kappa, M', b', i') = \lfloor v' \rfloor$ and $\alpha \vdash v \approx v'$.

- Commutation of stores to mapped blocks: if $\alpha \vdash M \approx M'$ and $\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')$ and $\alpha \vdash v \approx v'$ and $\texttt{store}(\kappa, M, b, i, v) = \lfloor M_1 \rfloor$, there exists M_1' such that $\texttt{store}(\kappa, M', b', i', v') = \lfloor M_1' \rfloor$ and $\alpha \vdash M_1 \approx M_1'$.
- Invariance by stores to unmapped blocks: if $\alpha \vdash M \approx M'$ and $\alpha(b) = \texttt{None}$ and $\texttt{store}(\kappa, M, b, i, v) = \lfloor M_1 \rfloor$, then $\alpha \vdash M_1 \approx M'$.

To enable the memory injection α to grow incrementally as new blocks are allocated during execution, we define the relation $\alpha' \geq \alpha$ (read: α' extends α) by $\forall b,\ \alpha'(b) = \alpha(b) \vee \alpha(b) = \texttt{None}$. The injection relations are preserved by extension of α. For instance, if $\alpha \vdash M \approx M'$, then $\alpha' \vdash M \approx M'$ for all α' such that $\alpha' \geq \alpha$.

4.2 Relating Execution Environments

Execution environments differ in structure between Clight and Cminor: the Clight environment E maps local variables to references of blocks containing the values of the variables, while in Cminor the environment E' for local variables map them directly to values. We define a matching relation $\mathit{EnvMatch}(\gamma, \alpha, E, M, E', sp)$ between a Clight environment E and memory state M and a Cminor environment E' and reference to a stack block sp as follows:

- For all variables x of type τ, if $\gamma(x) = \texttt{Local}$, then $\alpha(E(x)) = \texttt{None}$ and there exists v such that $\texttt{load}(\kappa(\tau), M, E(x), 0) = \lfloor v \rfloor$ and $\alpha \vdash v \approx E'(x)$.
- For all variables x of type τ, if $\gamma(x) = \texttt{Stack}(\delta)$, then $\alpha \vdash \texttt{Vptr}(E(x), 0) \approx \texttt{Vptr}(sp, \delta)$.
- For all $x \neq y$, we have $E(x) \neq E(y)$.
- If $\alpha(b) = \lfloor sp, \delta \rfloor$ for some block b and offset δ, then b is in the range of E.

The first two conditions express the preservation of the values of local variables during compilation. The last two rule out unwanted sharing between environment blocks and their images through α.

At any point during execution, several function calls may be active and we need to ensure matching between the environments of each call. For this, we introduce abstract call stacks, which are lists of 4-tuples (γ, E, E', sp) and record the environments of all active functions. A call stack cs is globally consistent with respect to C memory state M and memory injection α, written $\mathit{CallInv}(\alpha, M, cs)$, if $\mathit{EnvMatch}(\gamma, \alpha, E, M, E', sp)$ holds for all elements (γ, E, E', sp) of cs. Additional conditions, omitted for brevity, enforce separation between Clight environments E and between Cminor stack blocks sp belonging to different function activations in cs.

4.3 Proof by Simulation

The proof of semantic preservation for the translation proceeds by induction over the Clight evaluation derivation and case analysis on the last evaluation rule used. The proof shows that, assuming suitable consistency conditions over the

abstract call stack, the generated Cminor expressions and statements evaluate in ways that simulate the evaluation of the corresponding Clight expressions and statements.

We give a slightly simplified version of the simulation properties shown by induction over the Clight evaluation derivation. Let G' be the global Cminor environment obtained by translating all function definitions in the global Clight environment G. Assume $\mathit{CallInv}(\alpha, M, (\gamma, E, E', \mathit{sp}).\mathit{cs})$ and $\alpha \vdash M \approx M'$. Then there exists a Cminor environment E'_1, a Cminor memory state M'_1 and a memory injection $\alpha_1 \geq \alpha$ such that

- (R-values) If $G, E \vdash a, M \Rightarrow v, M_1$, there exists v' such that $G', \mathit{sp}, L \vdash \mathcal{R}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash v \approx v'$.
- (L-values) If $G, E \vdash a, M \stackrel{l}{\Rightarrow} \mathit{loc}, M_1$, there exists v' such that $G', \mathit{sp}, L \vdash \mathcal{L}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash \mathtt{Vptr}(\mathit{loc}) \approx v'$.
- (Statements) If $G, E \vdash s, M \Rightarrow \mathit{out}, M_1$ and τ_r is the return type of the function, there exists out' such that $G', \mathit{sp} \vdash \mathcal{S}_\gamma(s), E', M' \rightarrow \mathit{out}', E'_1, M'_1$ and $\alpha_1, \tau_r \vdash \mathit{out} \approx \mathit{out}'$.

Moreover, the final Clight and Cminor states satisfy $\alpha_1 \vdash M_1 \approx M'_1$ and $\mathit{CallInv}(\alpha_1, M_1, (\gamma, E, E'_1, \mathit{sp}).\mathit{cs})$.

In the case of statements, the relation between Clight and Cminor outcomes is defined as follows:

$$\alpha, \tau_r \vdash \mathtt{Out\_normal} \approx \mathtt{Out\_normal} \qquad \alpha, \tau_r \vdash \mathtt{Out\_continue} \approx \mathtt{Out\_exit}(0)$$
$$\alpha, \tau_r \vdash \mathtt{Out\_break} \approx \mathtt{Out\_exit}(1) \qquad \alpha, \tau_r \vdash \mathtt{Out\_return} \approx \mathtt{Out\_return}$$

$$\frac{\alpha \vdash \mathtt{cast}(v, \tau, \tau_r) \approx v'}{\alpha, \tau_r \vdash \mathtt{Out\_return}(v, \tau) \approx \mathtt{Out\_return}(v')}$$

In addition to the outer induction over the Clight evaluation derivation, the proofs proceed by copious case analysis, over the placement $\gamma(x)$ for accesses to variables x, and over the types of the operands for applications of overloaded operators. As a corollary of the simulation properties, we obtain the correctness theorem for the translation:

Theorem 1. *Assume the Clight program p is well-typed and translates without errors to a Cminor program p'. If $\vdash p \Rightarrow v$, and if v is an integer or float value, then $\vdash p' \rightarrow v$.*

This semantic preservation theorem applies only to terminating programs. Our choice of big-step operational semantics prevents us from reasoning over non-terminating executions.

The whole proof represents approximately 6000 lines of Coq statements and proof scripts, including 1000 lines (40 lemmas) for the properties of memory injections, 1400 lines (54 lemmas) for environment matching and the call stack invariant, 1400 lines (50 lemmas) for the translations of type-dependent operators and memory accesses, and 2000 lines (51 lemmas, one per Clight evaluation rule) for the final inductive proof of simulation. By comparison, the source code

of the Clight to Cminor translator is 800 lines of Coq function definitions. The proof is therefore 7.5 times bigger than the code it proves. The whole development (design and semantics of Clight; development of the translator; proof of its correctness) took approximately 8 person.months.

5 Related Work

Several formal semantics of C-like languages have been defined. Norrish [10] gives a small-step operational semantics, expressed using the HOL theorem prover, for a subset of C comparable to our Clight. His semantics captures exactly the non-determinism (partially unspecified evaluation order) allowed by the ISO C specification, making it significantly more complex than our deterministic semantics. Papaspyrou [11] addresses non-determinism as well, but using denotational semantics with monads. Abstract state machines have been used to give on-paper semantics for C [4,9] and more recently for C# [3].

Many correctness proofs of program transformations have been published, both on paper and machine-checked using proof assistants; see [2] for a survey. A representative example is [5], where a non-optimizing byte-code compiler from a subset of Java to a subset of the Java Virtual Machine is verified using Isabelle/HOL. Most of these correctness proofs apply to source languages that are either smaller or semantically cleaner than C.

The work that is closest to ours is part of the Verisoft project [6,12]. Using Isabelle/HOL, they formalize the semantics of C0 (a subset of the C language) and a compiler from C0 down to DLX assembly code. C0 is a type-safe subset of C, close to Pascal, and significantly smaller than our Clight: there is no pointer arithmetic, nor side effects, nor premature execution of statements and there exists only a single integer type, thus avoiding operator overloading. They provide both a big step semantics and a small step semantics for C0, the latter enabling reasoning about non-terminating and concurrent executions, unlike our big-step semantics. Their C0 compiler is a single pass compiler that generates unoptimized machine code. It is more complex than our translation from Clight to Cminor, but considerably simpler than our whole certified compiler.

6 Concluding Remarks

The C language is not pretty; this shows up in the relative complexity of our formal semantics and translation scheme. However, this complexity remains manageable with the tools (the Coq proof assistant) and the methodology (big-step semantics; simulation arguments; extraction of an executable compiler from its functional Coq specification) that we used.

Future work includes 1- handling a larger subset of C, especially `struct` types; and 2- evaluating the usability of the semantics for program proof and static analysis purposes. In particular, it would be interesting to develop axiomatic semantics (probably based on separation logic) for Clight and validate them against our operational semantics.

References

1. S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *Proc. of Int. Conf. on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 280–299, Manchester, UK, Nov. 2005. Springer-Verlag.
2. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
3. E. Börger, N. Fruja, V.Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2-3):235–284, 2005.
4. Y. Gurevich and J. Huggins. The semantics of the C programming language. In *Proc. of CSL'92 (Computer Science Logic)*, volume 702 of *LNCS*, pages 274–308. Springer Verlag, 1993.
5. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Mar. 2004. To appear in ACM TOPLAS.
6. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler. In *Proc. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 2–11, Koblenz, Germany, Sept. 2005. IEEE Computer Society Press.
7. X. Leroy. The Compcert certified compiler back-end – commented Coq development. Available on-line at `http://cristal.inria.fr/~xleroy`, 2006.
8. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. Symp. Principles Of Programming Languages (POPL)*, pages 42–54, Charleston, USA, Jan. 2006. ACM Press.
9. V. Nepomniaschy, I. Anureev, and A. Promsky. Verification-oriented language C-light and its structural operational semantics. In *Ershov Memorial Conference*, pages 103–111, 2003.
10. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, Dec. 1998.
11. N. Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, Feb. 1998.
12. M. Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, Apr. 2005.

J Autom Reasoning (2008) 41:1–31
DOI 10.1007/s10817-008-9099-0

Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations

Xavier Leroy · Sandrine Blazy

Received: 26 September 2007 / Accepted: 13 February 2008 / Published online: 14 March 2008

Abstract This article presents the formal verification, using the Coq proof assistant, of a memory model for low-level imperative languages such as C and compiler intermediate languages. Beyond giving semantics to pointer-based programs, this model supports reasoning over transformations of such programs. We show how the properties of the memory model are used to prove semantic preservation for three passes of the Compcert verified compiler.

Keywords Memory model · C · Program verification · Compilation · Compiler correctness · The Coq proof assistant

1 Introduction

A prerequisite to the formal verification of computer programs—by model checking, program proof, static analysis, or any other means—is to formalize the semantics of the programming language in which the program is written, in a way that is exploitable by the verification tools used. In the case of program proofs, these formal semantics are often presented in operational or axiomatic styles, e.g. Hoare logic. The need for formal semantics is even higher when the program being verified itself operates over programs: compilers, program analyzers, etc. In the case of a compiler, for instance, no less than three formal semantics are required: one for the implementation language of the compiler, one for the source language, and one for

X. Leroy (✉)
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
e-mail: Xavier.Leroy@inria.fr

S. Blazy
ENSIIE, 1 square de la Résistance, 91025 Evry cedex, France
e-mail: Sandrine.Blazy@ensiie.fr

the target language. More generally speaking, formal semantics "on machine" (that is, presented in a form that can be exploited by verification tools) are an important aspect of formal methods.

Formal semantics are relatively straightforward in the case of declarative programming languages. However, many programs that require formal verification are written in imperative languages featuring pointers (or references) and in-place modification of data structures. Giving semantics to these imperative constructs requires the development of an adequate *memory model*, that is, a formal description of the memory store and operations over it. The memory model is often a delicate part of a formal semantics for an imperative programming language. A very concrete memory model (e.g. representing the memory as a single array of bytes) can fail to validate algebraic laws over loads and stores that are actually valid in the programming language, making program proofs more difficult. An excessively abstract memory model can fail to account for e.g. aliasing or partial overlap between memory areas, thus causing the semantics to be incorrect.

This article reports on the formalization and verification, using the Coq proof assistant, of a memory model for C-like imperative languages. C and related languages are challenging from the standpoint of the memory model, because they feature both pointers and pointer arithmetic, on the one hand, and isolation and freshness guarantees on the other. For instance, pointer arithmetic can result in aliasing or partial overlap between the memory areas referenced by two pointers; yet, it is guaranteed that the memory areas corresponding to two distinct variables or two successive calls to `malloc` are disjoint. This stands in contrast with both higher-level imperative languages such as Java, where two distinct references always refer to disjoint areas, and lower-level languages such as machine code, where unrestricted address arithmetic invalidates all isolation guarantees.

The memory model presented here is used in the formal verification of the Compcert compiler [3, 15], a moderately-optimizing compiler that translates the Clight subset of the C programming language down to PowerPC assembly code. The memory model is used by the formal semantics of all languages manipulated by the compiler: the source language, the target language, and 7 intermediate languages that bridge the semantic gap between source and target. Certain passes of the compiler perform non-trivial transformations on memory allocations and accesses: for instance, local variables of a Clight function, initially mapped to individually-allocated memory blocks, are at some point mapped to sub-blocks of a single stack-allocated activation record, which at a later point is extended to make room for storing spilled temporaries. Proving the correctness (semantic preservation) of these transformations requires extensive reasoning over memory states, using the properties of the memory model given further in the paper.

The remainder of this article is organized as follows. Section 2 axiomatizes the values that are stored in memory states and the associated memory data types. Section 3 specifies an abstract memory model and illustrates its use for reasoning over programs. Section 4 defines the concrete implementation of the memory model used in Compcert and shows that it satisfies both the abstract specification and additional useful properties. Section 5 describes the transformations over memory states performed by three passes of the Compcert compiler. It then defines the memory invariants and proves the simulation results between memory operations that play a crucial role in proving semantics preservation for these three passes.

Section 6 briefly comments on the Coq mechanization of these results. Related work is discussed in Section 7, followed by conclusions and perspectives in Section 8.

All results presented in this article have been mechanically verified using the Coq proof assistant [2, 8]. The complete Coq development is available online at http://gallium.inria.fr/~xleroy/memory-model/. Consequently, the paper only sketches the proofs of some of its results; the reader is referred to the Coq development for the full proofs.

2 Values and Data Types

We assume given a set `val` of values, ranged over by v, used in the dynamic semantics of the languages to represent the results of calculations. In the Compcert development, `val` is defined as the discriminated union of 32-bit integers `int`(n), 64-bit double-precision floating-point numbers `float`(f), memory locations `ptr`(b, i) where b is a memory block reference b and i a byte offset within this block, and the constant `undef` representing an undefined value such as the value of an uninitialized variable.

We also assume given a set `memtype` of memory data types, ranged over by τ. Every memory access (`load` or `store` operation) takes as argument a memory data type, serving two purposes: (1) to indicate the size and natural alignment of the data being accessed, and (2) to enforce compatibility guarantees between the type with which a data was stored and the type with which it is read back. For a semantics for C, we can use C type expressions from the source language as memory data types. For the Compcert intermediate languages, we use the following set of memory data types, corresponding to the data that the target processor can access in one `load` or `store` instruction:

$\tau ::=$ `int8signed` | `int8unsigned`	8-bit integers
| `int16signed` | `int16unsigned`	16-bit integers
| `int32`	32-bit integers or pointers
| `float32`	32-bit, single-precision floats
| `float64`	64-bit, double-precision floats

The first role of a memory data type τ is to determine the size $|\tau|$ in bytes that a data of type τ occupies in memory, as well as the natural alignment $\langle\tau\rangle$ for data of this type. The alignment $\langle\tau\rangle$ models the address alignment constraints that many processors impose, e.g., the address of a 32-bit integer must be a multiple of 4. Both size and alignment are positive integers.[1,2]

(A1) $|\tau| > 0$ and $\langle\tau\rangle > 0$

[1] In this article, we write A for axioms, that is, assertions that we will not prove; S for specifications, that is, expected properties of the abstract memory model which the concrete model, as well as any other implementation, satisfies; D for derived properties, provable from the specifications; and P for properties of the concrete memory model.

[2] Throughout this article, variables occurring free in mathematical statements are implicitly universally quantified at the beginning of the statement.

To reason about some memory transformations, it is useful to assume that there exists a maximal alignment `max_alignment` that is a multiple of all possible alignment values:

(A2) $\langle\tau\rangle$ divides `max_alignment`

For the semantics of C, $|\tau|$ is the size of the type τ as returned by the `sizeof` operator of C. A possible choice for $\langle\tau\rangle$ is the size of the largest scalar type occurring in τ. For the Compcert intermediate languages, we take:

$$|\texttt{int8signed}| = |\texttt{int8unsigned}| = 1$$
$$|\texttt{int16signed}| = |\texttt{int16unsigned}| = 2$$
$$|\texttt{int32}| = |\texttt{float32}| = 4$$
$$|\texttt{float64}| = 8$$

Concerning alignments, Compcert takes $\langle\tau\rangle = 1$ and $\texttt{max\_alignment} = 1$, since the target architecture (PowerPC) has no alignment constraints. To model a target architecture with alignment constraints such as the Sparc, we would take $\langle\tau\rangle = |\tau|$ and $\texttt{max\_alignment} = 8$.

We now turn to the second role of memory data types, namely a form of dynamic type-checking. For a strongly-typed language, a memory state is simply a partial mapping from memory locations to values: either the language is statically typed, guaranteeing at compile-time that a value written with type τ is always read back with type τ; or the language is dynamically typed, in which case the generated machine code contains enough run-time type tests to enforce this property. However, the C language and most compiler intermediate languages are weakly typed. Consider a C "union" variable:

```
union { int i; float f; } u;
```

It is possible to assign an integer to `u.i`, then read it back as a float via `u.f`. This will not be detected at compile-time, and the C compiler will not generate code to prevent this. Yet, the C standard [13] specifies that this code has undefined behavior. More generally, after writing a data of type τ to a memory location, this location can only be read back with the same type τ or a compatible type; the behavior is undefined otherwise [13, Section 6.5, items 6 and 7]. To capture this behavior in a formal semantics for C, the memory state associates type-value pairs (τ, v), and not just values, to locations. Every `load` with type τ' at this location will check compatibility between the actual type τ of the location and the expected type τ', and fail if they are not compatible.

We abstract this notion of compatibility as a relation $\tau \sim \tau'$ between types. We assume that a type is always compatible with itself, and that compatible types have the same size and the same alignment:

(A3) $\tau \sim \tau$
(A4) If $\tau_1 \sim \tau_2$, then $|\tau_1| = |\tau_2|$ and $\langle\tau_1\rangle = \langle\tau_2\rangle$

Several definitions of the $\sim$ relation are possible, leading to different instantiations of our memory model. In the strictest instantiation,, $\tau \sim \tau'$ holds only if $\tau = \tau'$; that is, no implicit casts are allowed during a `store-load` sequence. The C standard

actually permits some such casts [13, Section 6.5, item 7]. For example, an integer n can be stored as an `unsigned char`, then reliably read back as a `signed char`, with result `(signed char)` n. This can be captured in our framework by stating that `unsigned char` $\sim$ `signed char`. For the Compcert intermediate languages, we go one step further and define $\tau_1 \sim \tau_2$ as $|\tau_1| = |\tau_2|$.

To interpret implicit casts in a `store-load` sequence, we need a function `convert` : `val` $\times$ `memtype` $\to$ `val` that performs these casts. More precisely, writing a value v with type τ, then reading it back with a compatible type τ' results in value `convert`(v, τ'). For the strict instantiation of the model, we take `convert`$(v, \tau') = v$. For the interpretation closest to the C standard, we need `convert`$(v, \tau') = (\tau')\ v$, where the right-hand side denotes a C type cast. Finally, for the Compcert intermediate languages, `convert` is defined as:

$$
\begin{aligned}
\texttt{convert}(\texttt{int}(n), \texttt{int8unsigned}) &= \texttt{int}(\text{8-bit zero extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int8signed}) &= \texttt{int}(\text{8-bit sign extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int16unsigned}) &= \texttt{int}(\text{16-bit zero extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int16signed}) &= \texttt{int}(\text{16-bit sign extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int32}) &= \texttt{int}(n)\\
\texttt{convert}(\texttt{ptr}(b, i), \texttt{int32}) &= \texttt{ptr}(b, i)\\
\texttt{convert}(\texttt{float}(f), \texttt{float32}) &= \texttt{float}(f \text{ normalized to single precision})\\
\texttt{convert}(\texttt{float}(f), \texttt{float64}) &= \texttt{float}(f)\\
\texttt{convert}(v, \tau) &= \texttt{undef} \quad \text{in all other cases}
\end{aligned}
$$

Note that this definition of `convert`, along with the fact that $\tau_1 \not\sim \tau_2$ if $|\tau_1| \neq |\tau_2|$, ensures that low-level implementation details such as the memory endianness or the bit-level representation of floats cannot be observed by Compcert intermediate programs. For instance, writing a float f with type `float32` and reading it back with compatible type `int32` results in the undefined value `undef` and not in the integer corresponding to the bit-pattern for f.

3 Abstract Memory Model

We now give an abstract, incomplete specification of a memory model that attempts to formalize the memory-related aspects of C and related languages. We have an abstract type `block` of references to memory blocks, and an abstract type `mem` of memory states. Intuitively, we view the memory state as a collection of separated blocks, identified by block references b. Each block behaves like an array of bytes, and is addressed using byte offsets $i \in \mathbb{Z}$. A memory location is therefore a pair (b, i) of a block reference b and an offset i within this block. The constant `empty` : `mem`

represents the initial memory state. Four operations over memory states are provided as total functions:

$$\begin{aligned}
\texttt{alloc} &: \texttt{mem} \times \mathbb{Z} \times \mathbb{Z} \to \texttt{option}(\texttt{block} \times \texttt{mem}) \\
\texttt{free} &: \texttt{mem} \times \texttt{block} \to \texttt{option}\ \texttt{mem} \\
\texttt{load} &: \texttt{memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \to \texttt{option}\ \texttt{val} \\
\texttt{store} &: \texttt{memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \times \texttt{val} \to \texttt{option}\ \texttt{mem}
\end{aligned}$$

Option types are used to represent potential failures. A value of type $\texttt{option}\,t$ is either ε (pronounced "none"), denoting failure, or $\lfloor x \rfloor$ (pronounced "some x"), denoting success with result $x : t$.

Allocation of a fresh memory block is written $\texttt{alloc}(m, l, h)$, where m is the initial memory state, and $l \in \mathbb{Z}$ and $h \in \mathbb{Z}$ are the low and high bounds for the fresh block. The allocated block has size $h - l$ bytes and can be accessed at byte offsets $l, l + 1, \ldots, h - 2, h - 1$. In other terms, the low bound l is inclusive but the high bound h is exclusive. Allocation can fail and return ε if not enough memory is available. Otherwise, $\lfloor b, m' \rfloor$ is returned, where b is the reference to the new block and m' the updated memory state.

Conversely, $\texttt{free}(m, b)$ deallocates block b in memory m. It can fail if e.g., b was already deallocated. In case of success, an updated memory state is returned.

Reading from memory is written $\texttt{load}(\tau, m, b, i)$. A data of type τ is read from block b of memory state m at byte offset i. If successful, the value thus read is returned. The memory state is unchanged.

Symmetrically, $\texttt{store}(\tau, m, b, i, v)$ writes value v at offset i in block b of m. If successful, the updated memory state is returned.

We now axiomatize the expected properties of these operations. The properties are labeled S to emphasize that they are specifications that any implementation of the model must satisfy. The first hypotheses are "good variable" properties defining the behavior of a `load` following an `alloc`, `free` or `store` operation.

(S5) If $\texttt{alloc}(m,l,h) = \lfloor b, m' \rfloor$ and $b' \neq b$, then $\texttt{load}(\tau, m', b', i) = \texttt{load}(\tau, m, b', i)$

(S6) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $\texttt{load}(\tau, m', b', i) = \texttt{load}(\tau, m, b', i)$

(S7) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $\tau \sim \tau'$, then $\texttt{load}(\tau', m', b, i) = \texttt{convert}(v, \tau')$

(S8) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $b' \neq b \ \vee\ i' + |\tau'| \leq i \ \vee\ i + |\tau| \leq i'$, then $\texttt{load}(\tau', m', b', i') = \texttt{load}(\tau', m, b', i')$

Hypotheses S5 and S6 state that allocating a block b or freeing a block b preserves loads performed in any other block $b' \neq b$. Hypothesis S7 states that after writing value v with type τ at offset i in block b, reading from the same location with a compatible type τ' succeeds and returns the value $\texttt{convert}(v, \tau')$. Hypothesis S8 states that storing a value of type τ in block b at offset i commutes with loading a value of type τ' in block b' at offset i', provided the memory areas corresponding to the `store` and the `load` are separate: either $b' \neq b$, or the range $[i, i + |\tau|)$ of byte offsets modified by the `store` is disjoint from the range $[i', i' + |\tau'|)$ read by the `load`.

Note that the properties above do not fully specify the `load` operation: nothing can be proved about the result of loading from a freshly allocated block, or freshly

deallocated block, or just after a store with a type and location that do not fall in the S7 and S8 case. This under-specification is intentional and follows the C standard. The concrete memory model of Section 4 will fully specify these behaviors.

The "good variable" properties use hypotheses $b' \neq b$, that is, separation properties between blocks. To establish such properties, we axiomatize the relation $m \models b$, meaning that the block reference b is valid in memory m. Intuitively, a block reference is valid if it was previously allocated but not yet deallocated; this is how the $m \models b$ relation will be defined in Section 4.

(S9) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $\neg(m \models b)$.
(S10) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m' \models b' \Leftrightarrow b' = b \vee m \models b'$
(S11) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \models b' \Leftrightarrow m \models b'$
(S12) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $m' \models b' \Leftrightarrow m \models b'$
(S13) If $m \models b$, then there exists m' such that $\texttt{free}(m, b) = \lfloor m' \rfloor$.

Hypothesis S9 says that the block returned by `alloc` is fresh, i.e., distinct from any other block that was valid in the initial memory state. Hypothesis S10 says that the newly allocated block is valid in the final memory state, as well as all blocks that were valid in the initial state. Block validity is preserved by `store` operations (S11). After a $\texttt{free}(m, b)$ operation, all initially valid blocks other than b remain valid, but it is unspecified whether the deallocated block b is valid or not (S12). Finally, the `free` operation is guaranteed to succeed when applied to a valid block (S13).

The next group of hypotheses axiomatizes the function $\mathcal{B}(m, b)$ that associates low and high bounds l, h to a block b in memory state m. We write $\mathcal{B}(m, b) = [l, h)$ to emphasize the meaning of bounds as semi-open intervals of allowed byte offsets within block b.

(S14) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $\mathcal{B}(m', b) = [l, h)$.
(S15) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $b' \neq b$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.
(S16) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.
(S17) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.

A freshly allocated block has the bounds that were given as argument to the `alloc` function (S14). The bounds of a block b' are preserved by an `alloc`, `store` or `free` operation, provided b' is not the block being allocated or deallocated.

For convenience, we write $\mathcal{L}(m, b)$ and $\mathcal{H}(m, b)$ for the low and high bounds attached to b, respectively, so that $\mathcal{B}(m, b) = [\mathcal{L}(m, b), \mathcal{H}(m, b))$.

Combining block validity with bound information, we define the "valid access" relation $m \models \tau \ @ \ b, i$, meaning that in state m, it is valid to write with type τ in block b at offset i.

$$m \models \tau \ @ \ b, i \stackrel{\text{def}}{=} m \models b \ \wedge \ \langle \tau \rangle \text{ divides } i \ \wedge \ \mathcal{L}(m, b) \leq i \ \wedge \ i + |\tau| \leq \mathcal{H}(m, b)$$

In other words, b is a valid block, the range $[i, i + |\tau|)$ of byte offsets being accessed is included in the bounds of b, and the offset i is an integer multiple of the alignment $\langle \tau \rangle$. If these conditions hold, we impose that the corresponding `store` operation succeeds.

(S18) If $m \models \tau \ @ \ b, i$ then there exists m' such that $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$.

Here are some derived properties of the valid access relation, easily provable from the hypotheses above.

(D19) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $\langle \tau \rangle$ divides i and $l \leq i$ and $i + |\tau| \leq h$, then $m' \models \tau \ @ \ b, i$.
(D20) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $m \models \tau \ @ \ b', i$, then $m' \models \tau \ @ \ b', i$.
(D21) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \models \tau \ @ \ b', i \ \Leftrightarrow \ m \models \tau \ @ \ b', i$.
(D22) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $m' \models \tau \ @ \ b', i \ \Leftrightarrow \ m \models \tau \ @ \ b', i$.

Proof D19 follows from S10 and S14. D20 follows from S10 and S15, noticing that $b' \neq b$ by S9. D21 follows from S11 and S16, and D22 from S12 and S17. □

To finish this section, we show by way of an example that the properties axiomatized above are sufficient to reason over the behavior of a C pointer program using axiomatic semantics. Consider the following C code fragment:

```
int * x = malloc(2 * sizeof(int));
int * y = malloc(sizeof(int));
x[0]    = 0;
x[1]    = 1;
*y      = x[0];
x[0]    = x[1];
x[1]    = *y;
```

We would like to show that in the final state, `x[0]` is 1 and `x[1]` is 0. Assuming that errors are automatically propagated using a monadic interpretation, we can represent the code fragment above as follows, using the operations of the memory model to make explicit memory operations. The variable `m` holds the current memory state. We also annotate the code with logical assertions expressed in terms of the memory model. The notation Γ stands for the three conditions $\texttt{x} \neq \texttt{y}$, $\texttt{m} \models \texttt{x}$, $\texttt{m} \models \texttt{y}$.

```
(x, m) = alloc(m, 0, 8);
         /* m ⊨ x */
(y, m) = alloc(m, 0, 4);
         /* Γ */
m = store(int, x, 0, 0);
         /* Γ, load(m, x, 0) = ⌊0⌋ */
m = store(int, x, 4, 1);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋ */
t = load(int, x, 0);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋, t = 0 */
m = store(int, y, 0, t);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋,
                                          load(m, y, 0) = ⌊0⌋ */
t = load(int, x, 4);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋,
                                  load(m, y, 0) = ⌊0⌋, t = 1 */
m = store(int, x, 0, t);
         /* Γ, load(m, x, 0) = ⌊1⌋, load(m, x, 4) = ⌊1⌋,
                                          load(m, y, 0) = ⌊0⌋ */
```

```
t = load(int, y, 0);
          /* Γ, load(m, x, 0) = ⌊1⌋, load(m, x, 4) = ⌊1⌋,
                                          load(m, y, 0) = ⌊0⌋, t = 0 */
m = store(int, x, 4, t);
          /* Γ, load(m, x, 0) = ⌊1⌋, load(m, x, 4) = ⌊0⌋,
                                          load(m, y, 0) = ⌊0⌋ */
```

Every postcondition can be proved from its precondition using the hypotheses listed in this section. The validity of blocks `x` and `y`, as well as the inequality $\texttt{x} \neq \texttt{y}$, follow from S9, S10 and S11. The assertions over the results of `load` operations and over the value of the temporary `t` follow from the good variable properties S7 and S8. Additionally, we can show that the `store` operations do not fail using S18 and the additional invariants $\texttt{m} \models \texttt{int} \mathbin{@} \texttt{x}, 0$ and $\texttt{m} \models \texttt{int} \mathbin{@} \texttt{x}, 4$ and $\texttt{m} \models \texttt{int} \mathbin{@} \texttt{y}, 0$, which follow from D19, D20 and D21.

4 Concrete Memory Model

We now develop a concrete implementation of a memory model that satisfies the axiomatization in Section 3. The type `block` of memory block references is implemented by the type $\mathbb{N}$ of nonnegative integers. Memory states (type `mem`) are quadruples (N, B, F, C), where

- $N : \texttt{block}$ is the first block not yet allocated;
- $B : \texttt{block} \to \mathbb{Z} \times \mathbb{Z}$ associates bounds to each block reference;
- $F : \texttt{block} \to \texttt{boolean}$ says, for each block, whether it has been deallocated (`true`) or not (`false`);
- $C : \texttt{block} \to \mathbb{Z} \to \texttt{option}\ (\texttt{memtype} \times \texttt{val})$ associates a content to each block b and each byte offset i. A content is either ε, meaning "invalid", or $\lfloor \tau, v \rfloor$, meaning that a value v was stored at this location with type τ.

We define block validity $m \models b$, where $m = (N, B, F, C)$, as $b < N \ \wedge \ F(b) = \texttt{false}$, that is, b was previously allocated ($b < N$) but not previously deallocated ($F(b) = \texttt{false}$). Similarly, the bounds $\mathcal{B}(m, b)$ are defined as $B(b)$.

The definitions of the constant `empty` and the operations `alloc`, `free`, `load` and `store` follow. We write $m = (N, B, F, C)$ for the initial memory state.

$$
\begin{aligned}
&\texttt{empty} = \\
&\quad (0,\ \lambda b.\,[0, 0),\ \lambda b.\,\texttt{false},\ \lambda b.\lambda i.\,\varepsilon) \\
&\texttt{alloc}(m, l, h) = \\
&\quad \text{if } \texttt{can\_allocate}(m, h - l) \text{ then } \lfloor b, m' \rfloor \text{ else } \varepsilon \\
&\quad \text{where } b = N \\
&\quad \text{and } m' = (N + 1, B\{b \leftarrow [l, h)\}, F\{b \leftarrow \texttt{false}\}, C\{b \leftarrow \lambda i.\,\varepsilon\}) \\
&\texttt{free}(m, b) = \\
&\quad \text{if not } m \models b \text{ then } \varepsilon \\
&\quad \text{else } \lfloor N, B\{b \leftarrow [0, 0)\}, F\{b \leftarrow \texttt{true}\}, C \rfloor \\
&\texttt{store}(\tau, m, b, i, v) = \\
&\quad \text{if not } m \models \tau \mathbin{@} b, i \text{ then } \varepsilon \\
&\quad \text{else } \lfloor N, B, F, C\{b \leftarrow c'\} \rfloor \\
&\quad \text{where } c' = C(b)\{i \leftarrow \lfloor \tau, v \rfloor, i + 1 \leftarrow \varepsilon, \ldots, i + |\tau| - 1 \leftarrow \varepsilon\}
\end{aligned}
$$

```
load(τ, m, b, i) =
  if not m ⊨ τ @ b, i then ε
  else if C(b)(i) = ⌊τ′, v⌋ and τ′ ∼ τ
      and C(b)(i + j) = ε for j = 1, ..., |τ| − 1
  then ⌊convert(v, τ)⌋
  else ⌊undef⌋
```

Allocation is performed by incrementing linearly the N component of the memory state. Block identifiers are never reused, which greatly facilitates reasoning over "dangling pointers" (references to blocks previously deallocated). The new block is given bounds $[l, h)$, deallocated status `false`, and invalid contents $\lambda i.\ \varepsilon$.[3] An unspecified, boolean-valued `can_allocate` function is used to model the possibility of failure if the request ($h - l$ bytes) exceeds the available memory. In the Compcert development, `can_allocate` always returns `true`, therefore modeling an infinite memory.

Freeing a block simply sets its deallocated status to `true`, rendering this block invalid, and its bounds to $[0, 0)$, reflecting the fact that this block no longer occupies any memory space.

A memory store first checks block and bounds validity using the $m \models \tau\ @\ b, i$ predicate, which is decidable. The contents $C(b)$ of block b are set to $\lfloor \tau, v \rfloor$ at offset i, recording the store done at this offset, and to ε at offsets $i + 1, \ldots, i + |\tau| - 1$, invalidating whatever data was previously stored at these addresses.

A memory load checks several conditions: first, that the block and offset being addressed are valid and within bounds; second, that block b at offset i contains a valid data $\lfloor v, \tau' \rfloor$; third, that the type τ' of this data is compatible with the requested type τ; fourth, that the contents of offsets $i + 1$ to $i + |\tau| - 1$ in block b are invalid, ensuring that the data previously stored at i in b was not partially overwritten by a store at an overlapping offset.

It is easy to show that this implementation satisfies the specifications given in Section 3.

Lemma 1 *Properties S5 to S18 are satisfied.*

Proof Most properties follow immediately from the definitions of `alloc`, `free`, `load` and `store` given above. For the "good variable" property S7, the `store` assigned contents $\lfloor \tau, v \rfloor, \varepsilon, \ldots, \varepsilon$ to offsets $i, \ldots, i + |\tau| - 1$, respectively. Since $|\tau'| = |\tau|$ by A1, the checks performed by `load` succeed. For the other "good variable" property, S8, the assignments performed by the `store` over $C(b)$ at offsets $i, \ldots, i + |\tau| - 1$ have no effect over the values of offsets $i', \ldots, i' + |\tau'| - 1$ in $C(b')$, given the separation hypothesis $(b' \neq b \ \vee\ i' + |\tau'| \leq i \ \vee\ i + |\tau| \leq i')$. □

Moreover, the implementation also enjoys a number of properties that we now state. In the following sections, we will only use these properties along with those

[3] Since blocks are never reused, the freshly-allocated block b already has deallocated status `false` and contents $\lambda i.\ \varepsilon$ in the initial memory state (N, B, F, C). Therefore, in the definition of `alloc`, the updates $F\{b \leftarrow \texttt{false}\}$ and $C\{b \leftarrow \lambda i.\ \varepsilon\}$ are not strictly necessary. However, they allow for simpler reasoning over the `alloc` function, making it unnecessary to prove the invariants $F(b) = \texttt{false}$ and $C(b) = \lambda i.\ \varepsilon$ for all $b \geq N$.

of Section 3, but not the precise definitions of the memory operations. The first two properties state that a `store` or a `load` succeeds if and only if the corresponding memory reference is valid.

(P24) $m \models \tau \ @ \ b, i \ \Leftrightarrow \ \exists m', \texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$
(P25) $m \models \tau \ @ \ b, i \ \Leftrightarrow \ \exists v, \texttt{load}(\tau, m, b, i) = \lfloor v \rfloor$

Then come additional properties capturing the behavior of a `load` following an `alloc` or a `store`. In circumstances where the "good variable" properties of the abstract memory model leave unspecified the result of the `load`, these "not-so-good variable" properties guarantee that the `load` predictably returns $\lfloor \texttt{undef} \rfloor$.

(P26) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $\texttt{load}(\tau, m', b, i) = \lfloor v \rfloor$, then $v = \texttt{undef}$.
(P27) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $\tau \not\sim \tau'$ and $\texttt{load}(\tau', m', b, i) = \lfloor v' \rfloor$, then $v' = \texttt{undef}$.
(P28) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$ and $\texttt{load}(\tau', m', b, i') = \lfloor v' \rfloor$, then $v' = \texttt{undef}$.

Proof For P26, the contents of m' at b, i are ε and therefore not of the form $\lfloor \tau, v \rfloor$. For P27, the test $\tau \sim \tau'$ in the definition of `load` fails. For P28, consider the contents c of block b in m'. If $i < i'$, the `store` set $c(i') = \varepsilon$. If $i > i'$, the `store` set $c(i' + j) = \lfloor \tau, v \rfloor$ for some $j \in [1, |\tau'|)$. In both cases, one of the checks in the definition of `load` fails. □

Combining properties S7, S8, P25, P27 and P28, we obtain a complete characterization of the behavior of a `load` that follows a `store`. (See Fig. 1 for a graphical illustration of the cases.)

(D29) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $m \models \tau' \ @ \ b', i'$, then one and only one of the following four cases holds:

- Compatible: $b' = b$ and $i' = i$ and $\tau \sim \tau'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{convert}(v, \tau') \rfloor$.
- Incompatible: $b' = b$ and $i' = i$ and $\tau \not\sim \tau'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{undef} \rfloor$.
- Disjoint: $b' \neq b$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$, in which case $\texttt{load}(\tau', m', b', i') = \texttt{load}(\tau', m, b', i')$.
- Overlapping: $b' = b$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{undef} \rfloor$.

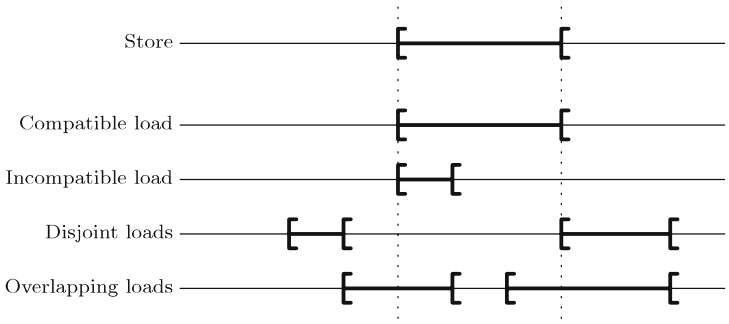


Fig. 1 A `store` followed by a `load` in the same block: the four cases of property D29

As previously mentioned, an interesting property of the concrete memory model is that `alloc` never reuses block identifiers, even if some blocks have been deallocated before. To account for this feature, we define the relation $m \# b$, pronounced "block b is fresh in memory m", and defined as $b \geq N$ if $m = (N, B, F, C)$. This relation enjoys the following properties:

(P30) $m \# b$ and $m \models b$ are mutually exclusive.
(P31) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m \# b$.
(P32) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m' \# b' \Leftrightarrow b' \neq b \wedge m \# b'$.
(P33) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \# b' \Leftrightarrow m \# b'$.
(P34) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $m' \# b' \Leftrightarrow m \# b'$.

Using the freshness relation, we say that two memory states m_1 and m_2 have the same domain, and write $\mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$, if $\forall b, (m_1 \# b \Leftrightarrow m_2 \# b)$. In our concrete implementation, two memory states have the same domain if and only if their N components are equal. Therefore, `alloc` is deterministic with respect to the domain of the current memory state: `alloc` chooses the same free block when applied twice to memory states that have the same domain, but may differ in block contents.

(P35) If $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ and $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$ and $\mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$, then $b_1 = b_2$ and $\mathrm{Dom}(m_1') = \mathrm{Dom}(m_2')$.

The last property of the concrete implementation used in the remainder of this paper is the following: a block b that has been deallocated is both invalid and empty, in the sense that its low and high bounds are equal.

(P36) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $\neg(m' \models b)$.
(P37) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $\mathcal{L}(m', b) = \mathcal{H}(m', b)$.

5 Memory Transformations

We now study the use of the concrete memory model to prove the correctness of program transformations as performed by compiler passes. Most passes of the Compcert compiler preserve the memory behavior of the program: some modify the flow of control, others modify the flow of data not stored in memory, but the memory states before and after program transformation match at every step of the program execution. The correctness proofs for these passes exploit none of the properties of the memory model. However, three passes of the Compcert compiler change the memory behavior of the program, and necessitate extensive reasoning over memory states to be proved correct. We now outline the transformations performed by these three passes.

The first pass that modifies the memory behavior is the translation from the source language Clight to the intermediate language Cminor. In Clight, all variables are allocated in memory: the evaluation environment maps variables to references of memory blocks that contain the current values of the variables. This is consistent with the C specification and the fact that the address of any variable can be taken and used as a memory pointer using the `&` operator. However, this feature renders register allocation and most other optimizations very difficult, because aliasing between a pointer and a variable is always possible.

Therefore, the Clight to Cminor translation detects scalar local variables whose address is never taken with the `&` operator, and "pulls them out of memory": they become Cminor local variables, whose current values are recorded in an environment separate from the memory state, and whose address cannot be taken. Other Clight local variables remain memory-allocated, but are grouped as sub-areas of a single memory block, the Cminor stack block, which is automatically allocated at function entry and deallocated at function exit. (See Fig. 2, left.)

Consequently, the memory behavior of the source Clight program and the transformed Cminor program differ greatly: when the Clight program allocates N fresh blocks at function entry for its N local variables, the Cminor program allocates only one; the `load` and `store` operations performed by the Clight semantics every time a local variable is accessed either disappear in Cminor or becomes `load` and `store` in sub-areas of the Cminor stack block.

The second pass that affects memory behavior is register allocation. In RTL, the source language for this translation, local variables and temporaries are initialized to the `undef` value on function entry. (This initialization agrees with the semantics of Clight, where reading an uninitialized local variable amounts to loading from a freshly allocated block.) After register allocation, some of these RTL variables and temporaries are mapped to global hardware registers, which are not initialized to the `undef` value on function entry, but instead keep whatever value they had in the caller function at point of call. This does not change the semantics of well-defined RTL programs, since the RTL semantics goes wrong whenever an `undef` value is involved in an arithmetic operation or conditional test. Therefore, values of uninitialized RTL variables do not participate in the computations performed by the program, and can be changed from `undef` to any other value without changing the semantics of the program. However, the original RTL program could have stored these values of uninitialized variables in memory locations. Therefore, the memory states before and after register allocation have the same shapes, but the contents of some memory locations can change from `undef` to any value, as pictured in Fig. 2, center.

The third and last pass where memory states differ between the original and transformed codes is the spilling pass performed after register allocation. Variables and temporaries that could not be allocated to hardware registers must be "spilled" to memory, that is, stored in locations within the stack frame of the current function. Additional stack frame space is also needed to save the values of callee-save registers

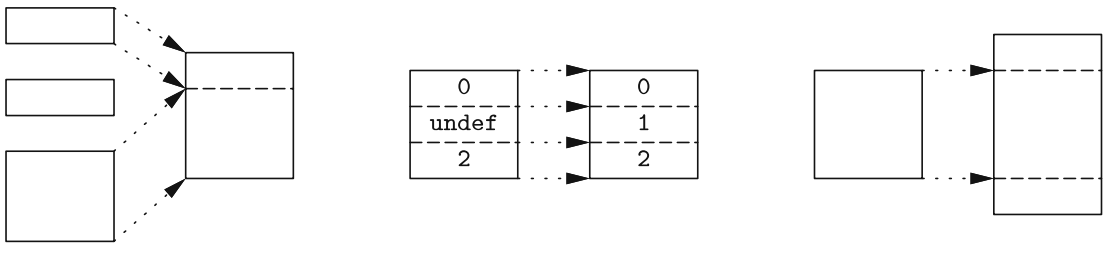


Fig. 2 Transformations over memory states in the Compcert compiler

on function entry. Therefore, the spilling pass needs to enlarge the stack frame that was laid out at the time of Cminor generation, to make room for spilled variables and saved registers. The memory state after spilling therefore differs from the state before spilling: stack frame blocks are larger, and the transformed program performs additional `load` and `store` operations to access spilled variables. (See Fig. 2, right.)

In the three examples of program transformations outlined above, we need to formalize an invariant that relates the memory states at every point of the executions of the original and transformed programs, and prove appropriate simulation results between the memory operations performed by the two programs. Three such relations between memory states are studied in the remainder of this section: memory extensions in Section 5.2, corresponding to the spilling pass; refinement of stored values in Section 5.3, corresponding to the register allocation pass; and memory injections in Section 5.4, corresponding to the Cminor generation pass. These three relations share a common basis, the notion of memory embeddings, defined and studied first in Section 5.1.

5.1 Generic Memory Embeddings

An *embedding* E is a function of type $\texttt{block} \to \texttt{option}(\texttt{block} \times \mathbb{Z})$ that establishes a correspondence between blocks of a memory state m_1 of the original program and blocks of a memory state m_2 of the transformed program. Let b_1 be a block reference in m_1. If $E(b_1) = \varepsilon$, this block corresponds to no block in m_2: it has been eliminated by the transformation. If $E(b_1) = \lfloor b_2, \delta \rfloor$, the block b_1 in m_1 corresponds to the block b_2 in m_2, or a sub-block thereof, with offsets being shifted by δ. That is, the memory location (b_1, i) in m_1 is associated to the location $(b_2, i + \delta)$ in m_2. We say that a block b of m_1 is *unmapped* in m_2 if $E(b) = \varepsilon$, and *mapped* otherwise.

We assume we are given a relation $E \vdash v_1 \hookrightarrow v_2$ between values v_1 of the original program and values v_2 of the transformed program, possibly parametrized by E. (Sections 5.2, 5.3 and 5.4 will particularize this relation).

We say that E embeds memory state m_1 in memory state m_2, and write $E \vdash m_1 \hookrightarrow m_2$, if every successful load from a mapped block of m_1 is simulated by a successful load from the corresponding sub-block in m_2, in the following sense:

$$\begin{aligned} E(b_1) \;=\; \lfloor b_2, \delta \rfloor \;\wedge\; \texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor \\ \Rightarrow\; \exists v_2, \texttt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor \;\wedge\; E \vdash v_1 \hookrightarrow v_2 \end{aligned}$$

We now state and prove commutation and simulation properties between the memory embedding relation and the operations of the concrete memory model. First, validity of accesses is preserved, in the following sense.

Lemma 2 *If $E(b_1) = \lfloor b_2, \delta \rfloor$ and $E \vdash m_1 \hookrightarrow m_2$, then $m_1 \models \tau \mathbin{@} b_1, i$ implies $m_2 \models \tau \mathbin{@} b_2, i + \delta$.*

Proof By property P25, there exists v_1 such that $\texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists v_2 such that $\texttt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor$. The result follows from property P25. □

When is the memory embedding relation preserved by memory stores? There are three cases to consider, depicted in Fig. 3. In the leftmost case, the original

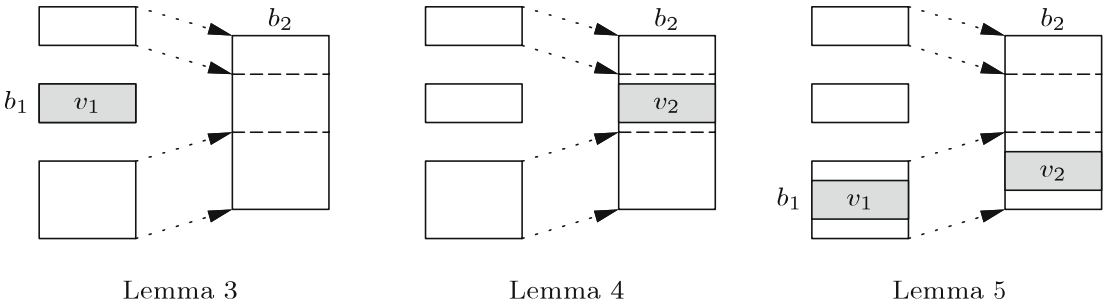


Fig. 3 The three simulation lemmas for memory stores. The *grayed areas* represent the locations of the stores. v_1 is a value stored by the original program and v_2 a value stored by the transformed program

program performs a store in memory m_1 within a block that is not mapped, while the transformed program performs no store, keeping its memory m_2 unchanged.

Lemma 3 *If* $E(b_1) = \varepsilon$ *and* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{store}(\tau, m_1, b_1, i, v) = \lfloor m_1' \rfloor$, *then* $E \vdash m_1' \hookrightarrow m_2$.

Proof Consider a load in m_1' from a mapped block: $E(b_1') = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau', m_1', b_1', i') = \lfloor v_1 \rfloor$. By hypothesis $E(b_1) = \varepsilon$, we have $b_1' \neq b_1$. By S8, it follows that $\texttt{load}(\tau', m_1, b_1', i') = \texttt{load}(\tau', m_1', b_1', i') = \lfloor v_1 \rfloor$. The result follows from hypothesis $E \vdash m_1 \hookrightarrow m_2$. □

In the second case (Fig. 3, center), the original program performs no store in its memory m_1, but the transformed program stores some data in an area of its memory m_2 that is disjoint from the images of the blocks of m_1.

Lemma 4 *Let* b_2, i, τ *be a memory reference in* m_2 *such that*

$$\forall b_1, \delta, \quad E(b_1) = \lfloor b_2, \delta \rfloor \;\Rightarrow\; \mathcal{H}(m_1, b_1) + \delta \leq i \;\vee\; i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$$

If $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{store}(\tau, m_2, b_2, i, v) = \lfloor m_2' \rfloor$, *then* $E \vdash m_1 \hookrightarrow m_2'$.

Proof Consider a load in m_1 from a mapped block: $E(b_1) = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau', m_1, b_1, i') = \lfloor v_1 \rfloor$. By P25, this load is within bounds: $\mathcal{L}(m_1, b_1) \leq i'$ and $i' + |\tau'| \leq \mathcal{H}(m_1, b_1)$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists v_2 such that $\texttt{load}(\tau, m_2, b_2', i' + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$. We check that the separation condition of S8 holds. This is obvious if $b_2' \neq b_2$. Otherwise, by hypothesis on b_2, either $\mathcal{H}(m_1, b_1) + \delta \leq i$ or $i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$. In the first case, $i' + \delta + |\tau'| \leq \mathcal{H}(m_1, b_1) + \delta \leq i$, and in the second case, $i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta \leq i' + \delta$. Therefore, $\texttt{load}(\tau', m_2', b_2', i' + \delta) = \texttt{load}(\tau', m_2, b_2', i' + \delta) = \lfloor v_2 \rfloor$, and the desired result follows. □

In the third case (Fig. 3, right), the original program stores a value v_1 in a mapped block of m_1, while in parallel the transformed program stores a matching value v_2 at the corresponding location in m_2. For this operation to preserve the memory embedding relation, it is necessary that the embedding E is *nonaliasing*. We say that

an embedding E is nonaliasing in a memory state m if distinct blocks are mapped to disjoint sub-blocks:

$$\begin{aligned} b_1 \neq b_2 \wedge E(b_1) = \lfloor b'_1, \delta_1 \rfloor \wedge E(b_2) = \lfloor b'_2, \delta_2 \rfloor \\ \Rightarrow b'_1 \neq b'_2 \\ \vee [\mathcal{L}(m, b_1) + \delta_1, \mathcal{H}(m, b_1) + \delta_1) \cap [\mathcal{L}(m, b_2) + \delta_2, \mathcal{H}(m, b_2) + \delta_2) = \emptyset \end{aligned}$$

The disjointness condition between the two intervals can be decomposed as follows: either $\mathcal{L}(m, b_1) \geq \mathcal{H}(m, b_1)$ (block b_1 is empty), or $\mathcal{L}(m, b_2) \geq \mathcal{H}(m, b_2)$ (block b_2 is empty), or $\mathcal{H}(m, b_1) + \delta_1 \leq \mathcal{L}(m, b_2) + \delta_2$, or $\mathcal{H}(m, b_2) + \delta_2 \leq \mathcal{L}(m, b_1) + \delta_1$.

Lemma 5 *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. *Let* v_1, v_2 *be two values and* τ *a type such that* v_1 *embeds in* v_2 *after conversion to any type* τ' *compatible with* τ*:*

$$\forall \tau', \ \tau \sim \tau' \Rightarrow E \vdash \texttt{convert}(v_1, \tau') \hookrightarrow \texttt{convert}(v_2, \tau')$$

If $E \vdash m_1 \hookrightarrow m_2$ *and* E *is nonaliasing in the memory state* m_1 *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $\texttt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m'_1 \rfloor$, *then there exists a memory state* m'_2 *such that* $\texttt{store}(\tau, m_2, b_2, i + \delta, v_2) = \lfloor m'_2 \rfloor$ *and moreover* $E \vdash m'_1 \hookrightarrow m'_2$.

Proof The existence of m'_2 follows from Lemma 2 and property P24. Consider a load in m'_1 from a mapped block: $E(b'_1) = \lfloor b'_2, \delta' \rfloor$ and $\texttt{load}(\tau', m'_1, b'_1, i') = \lfloor v'_1 \rfloor$. By property D29, there are four cases to consider.

- Compatible: $b'_1 = b_1$ and $i' = i$ and $\tau \sim \tau'$. In this case, $v'_1 = \texttt{convert}(v_1, \tau')$. By S7, we have $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \texttt{load}(\tau', m'_2, b_2, i + \delta) = \lfloor \texttt{convert}(v_2, \tau') \rfloor$. The result $E \vdash v'_1 \hookrightarrow \texttt{convert}(v_2, \tau')$ follows from the hypothesis over v_1 and v_2.
- Incompatible: $b'_1 = b_1$ and $i' = i$ and $\tau \not\sim \tau'$. In this case, $v'_1 = \texttt{undef}$. By P27 and P25, we have $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \texttt{load}(\tau', m'_2, b_2, i + \delta) = \lfloor \texttt{undef} \rfloor$. The result follows from the hypothesis $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$.
- Disjoint: $b'_1 \neq b_1$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$. In this case, $\texttt{load}(\tau', m_1, b'_1, i') = \lfloor v'_1 \rfloor$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists v'_2 such that $\texttt{load}(\tau', m_2, b'_2, i' + \delta') = \lfloor v'_2 \rfloor$ and $E \vdash v'_1 \hookrightarrow v'_2$. Exploiting the nonaliasing hypothesis over E and m_1, we show that the separation hypotheses of property S8 hold, which entails $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \lfloor v'_2 \rfloor$ and the expected result.
- Overlapping: $b'_1 = b_1$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$. In this case $v'_1 = \texttt{undef}$. We show $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \lfloor \texttt{undef} \rfloor$ using P28, and conclude using the hypothesis $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. □

We now turn to relating allocations with memory embeddings, starting with the case where two allocations are performed in parallel, one in the original program, the other in the transformed program.

Lemma 6 *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l_1, h_1) = \lfloor b_1, m'_1 \rfloor$ *and* $\texttt{alloc}(m_2, l_2, h_2) = \lfloor b_2, m'_2 \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $l_2 \leq l_1 + \delta$ *and* $h_1 + \delta \leq h_2$ *and* `max_alignment` *divides* δ, *then* $E \vdash m'_1 \hookrightarrow m'_2$.

Proof Consider a load in m'_1 from a mapped block: $E(b'_1) = \lfloor b'_2, \delta \rfloor$ and $\texttt{load}(\tau, m'_1, b'_1, i) = \lfloor v_1 \rfloor$. If $b'_1 \neq b_1$, we have $\texttt{load}(\tau, m_1, b'_1, i) = \lfloor v_1 \rfloor$ by S5, and there exists v_2 such that $\texttt{load}(\tau, m_2, b'_2, i + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$. It must be the case that $b'_2 \neq b_2$, otherwise the latter `load` would have failed (by S9 and P25). The expected result $\texttt{load}(\tau, m'_2, b'_2, i + \delta) = \lfloor v_2 \rfloor$ follows from S5.

If $b'_1 = b_1$, we have $\texttt{load}(\tau, m'_1, b_1, i) = \lfloor \texttt{undef} \rfloor$ by P26, and $l_1 \leq i, i + |\tau| \leq h_1$ and $|\tau|$ divides i by P25 and P26. It follows that $m'_2 \models \tau \ @ \ b_2, i + \delta$, and therefore $\texttt{load}(\tau, m'_2, b_2, i + \delta) = \lfloor \texttt{undef} \rfloor$ by P25 and P26. This is the expected result since $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. □

To complement Lemma 6, we also consider the cases where allocations are performed either in the original program or in the transformed program, but not necessarily in both. (See Fig. 4.) We omit the proof sketches, as they are similar to that of Lemma 6.

Lemma 7 *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m'_2 \rfloor$, *then* $E \vdash m_1 \hookrightarrow m'_2$.

Lemma 8 *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m'_1 \rfloor$ *and* $E(b_1) = \varepsilon$, *then* $E \vdash m'_1 \hookrightarrow m_2$.

Lemma 9 *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{v}$ *for all values* v. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m'_1 \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $m_2 \models b_2$ *and* $\mathcal{L}(m_2, b_2) \leq l + \delta$ *and* $h + \delta \leq \mathcal{H}(m_2, b_2)$ *and* `max_alignment` *divides* δ, *then* $E \vdash m'_1 \hookrightarrow m_2$.

Finally, we consider the interaction between `free` operations and memory embeddings. Deallocating a block in the original program always preserves embedding.

Lemma 10 *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_1, b_1) = \lfloor m'_1 \rfloor$, *then* $E \vdash m'_1 \hookrightarrow m_2$.

Proof If $\texttt{load}(\tau, m'_1, b'_1, i) = \lfloor v_1 \rfloor$, it must be that $b'_1 \neq b_1$ by P25 and P36. We then have $\texttt{load}(\tau, m_1, b'_1, i) = \lfloor v_1 \rfloor$ by S6 and conclude by hypothesis $E \vdash m_1 \hookrightarrow m_2$. □

Deallocating a block in the transformed program preserves embedding if no valid block of the original program is mapped to the deallocated block.

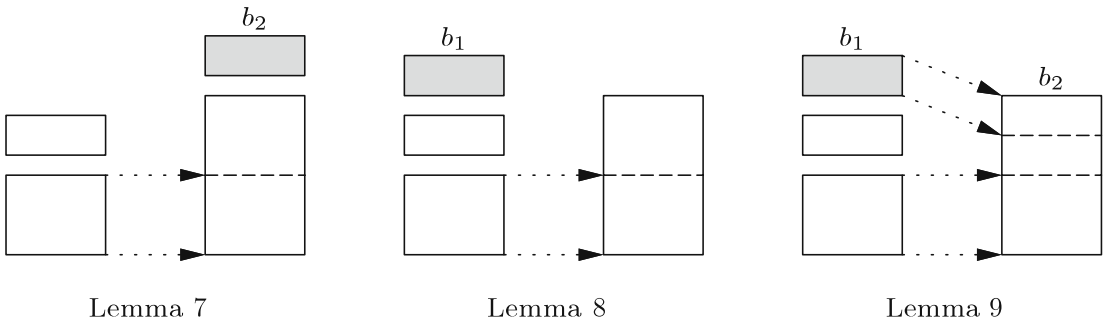


Fig. 4 The three simulation lemmas for memory allocations. The *grayed areas* represent the freshly allocated blocks

Lemma 11 *Assume* $\forall b_1, \delta, \quad E(b_1) = \lfloor b_2, \delta \rfloor \Rightarrow \neg(m_1 \models b_1)$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$, *then* $E \vdash m_1 \hookrightarrow m_2'$.

Proof Assume $E(b_1) = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$. It must be the case that $b_2' \neq b_2$, otherwise $m_1 \models b_1$ would not hold, contradicting P25. The result follows from the hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property S6. □

Combining Lemmas 10 and 11, we see that embedding is preserved by freeing a block b_1 in the original program and in parallel freeing a block b_2 in the transformed program, provided that no block other than b_1 maps to b_2.

Lemma 12 *Assume* $\forall b, \delta, E(b) = \lfloor b_2, \delta \rfloor \Rightarrow b = b_1$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_1, b_1) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$, *then* $E \vdash m_1' \hookrightarrow m_2'$.

Finally, it is useful to notice that the nonaliasing property of embeddings is preserved by `free` operations.

Lemma 13 *If* E *is nonaliasing in* m_1, *and* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$, *then* E *is nonaliasing in* m_1'.

Proof The block b becomes empty in m_1': by P37, $\mathcal{L}(m_1', b) = \mathcal{H}(m_1', b)$. The result follows from the definition of nonaliasing embeddings. □

5.2 Memory Extensions

We now instantiate the generic framework of Section 5.1 to account for the memory transformations performed by the spilling pass: the transformed program allocates larger blocks than the original program, and uses the extra space to store data of its own (right part of Fig. 2).

Figure 5 illustrates the effect of this transformation on the memory operations performed by the original and transformed programs. Each `alloc` in the original program becomes an `alloc` operation with possibly larger bounds. Each `store`, `load` and `free` in the original program corresponds to an identical operation in the transformed program, with the same arguments and results. The transformed program contains additional `store` and `load` operations, corresponding to spills

```
sp := alloc(0,8)
store(int, sp, 0, 42)
x := load(int, sp, 0)
...
...
...
y := x + x
free(sp)
```

```
sp := alloc(-4,8)
store(int, sp, 0, 42)
x := load(int, sp, 0)
store(int, sp, -4, x)   // spill
...
x := load(int, sp, -4)  // reload
y := x + x
free(sp)
```

Fig. 5 Example of insertion of spill code. *Left*: original program, *right*: transformed program. The variable `x` was spilled to the stack location at offset −4. The variable `y` was not spilled

and reloads of variables, operating on memory areas that were not accessible in the original program (here, the word at offset -4 in the block `sp`).

To prove that this transformation preserves semantics, we need a relation between the memory states of the original and transformed programs that (1) guarantees that matching pairs of `load` operations return the same value, and (2) is preserved by `alloc`, `store` and `free` operations.

In this section, we consider a fixed embedding E_{id} that is the identity function: $E_{id}(b) = \lfloor b, 0 \rfloor$ for all blocks b. Likewise, we define embedding between values as equality between these values: $E_{id} \vdash v_1 \hookrightarrow v_2$ if and only if $v_1 = v_2$.

We say that a transformed memory state m_2 extends an original memory state m_1, and write $m_1 \subseteq m_2$, if E_{id} embeds m_1 in m_2, and both memory states have the same domain:

$$m_1 \subseteq m_2 \quad \stackrel{\text{def}}{=} \quad E_{id} \vdash m_1 \hookrightarrow m_2 \ \wedge \ \mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$$

The $\subseteq$ relation over memory states is reflexive and transitive. It implies the desired equality between the results of a `load` performed by the initial program and the corresponding `load` after transformation.

Lemma 14 *If* $m_1 \subseteq m_2$ *and* $\texttt{load}(\tau, m_1, b, i) = \lfloor v \rfloor$, *then* $\texttt{load}(\tau, m_2, b, i) = \lfloor v \rfloor$.

Proof Since $E_{id} \vdash m_1 \hookrightarrow m_2$ holds, and $E_{id}(b) = \lfloor b, 0 \rfloor$, there exists a value v' such that $\texttt{load}(\tau, m_2, b, i) = \lfloor v' \rfloor$ and $E_{id} \vdash v \hookrightarrow v'$. The latter entails $v' = v$ and the expected result. □

We now show that any `alloc`, `store` or `free` operation over m_1 is simulated by a similar memory operation over m_2, preserving the memory extension relation.

Lemma 15 *Assume* $\texttt{alloc}(m_1, l_1, h_1) = \lfloor b_1, m_1' \rfloor$ *and* $\texttt{alloc}(m_2, l_2, h_2) = \lfloor b_2, m_2' \rfloor$. *If* $m_1 \subseteq m_2$ *and* $l_2 \leq l_1$ *and* $h_1 \leq h_2$, *then* $b_1 = b_2$ *and* $m_1' \subseteq m_2'$.

Proof The equality $b_1 = b_2$ follows from P35. The embedding $E_{id} \vdash m_1' \hookrightarrow m_2'$ follows from Lemma 6. The domain equality $\mathrm{Dom}(m_1') = \mathrm{Dom}(m_2')$ follows from P32. □

Lemma 16 *Assume* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b) = \lfloor m_2' \rfloor$. *If* $m_1 \subseteq m_2$, *then* $m_1' \subseteq m_2'$.

Proof Follows from Lemma 12 and property P34. □

Lemma 17 *If* $m_1 \subseteq m_2$ *and* $\texttt{store}(\tau, m_1, b, i, v) = \lfloor m_1' \rfloor$, *then there exists* m_2' *such that* $\texttt{store}(\tau, m_2, b, i, v) = \lfloor m_2' \rfloor$ *and* $m_1' \subseteq m_2'$.

Proof Follows from Lemma 5 and property P33. By construction, the embedding E_{id} is nonaliasing for any memory state. □

Finally, the transformed program can also perform additional stores, provided they fall outside the memory bounds of the original program. (These stores take place when a variable is spilled to memory.) Such stores preserve the extension relation.

Lemma 18 *Assume* $m_1 \subseteq m_2$ *and* $\texttt{store}(\tau, m_2, b, i, v) = \lfloor m_2' \rfloor$. *If* $i + |\tau| \leq \mathcal{L}(m_1, b)$ *or* $\mathcal{H}(m_1, b) \leq i$, *then* $m_1 \subseteq m_2'$.

Proof Follows from Lemma 4 and property P33. □

5.3 Refinement of Stored Values

In this section, we consider the case where the original and transformed programs allocate identically-sized blocks in lockstep, but some of the `undef` values produced and stored by the original program can be replaced by more defined values in the transformed program. This situation, depicted in the center of Fig. 2, occurs when verifying the register allocation pass of Compcert. Figure 6 outlines an example of this transformation.

We say that a value v_1 is refined by a value v_2, and we write $v_1 \leq v_2$, if either $v_1 = \texttt{undef}$ or $v_1 = v_2$. We assume that the `convert` function is compatible with refinements: $v_1 \leq v_2 \Rightarrow \texttt{convert}(v_1, \tau) \leq \texttt{convert}(v_2, \tau)$. (This is clearly the case for the examples of `convert` functions given at the end of Section 2.)

We instantiate again the generic framework of Section 5.1, using the identity embedding $E_{id} = \lambda b\,.\,\lfloor b, 0 \rfloor$ and the value embedding

$$E_{id} \vdash v_1 \hookrightarrow v_2 \quad \stackrel{\text{def}}{=} \quad v_1 \leq v_2.$$

We say that a transformed memory state m_2 refines an original memory state m_1, and write $m_1 \leq m_2$, if E_{id} embeds m_1 in m_2, and both memory states have the same domain:

$$m_1 \leq m_2 \quad \stackrel{\text{def}}{=} \quad E_{id} \vdash m_1 \hookrightarrow m_2 \ \wedge\ \mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$$

The $\leq$ relation over memory states is reflexive and transitive.

The following simulation properties are immediate consequences of the results from Section 5.1.

Lemma 19 *Assume* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$. *If* $m_1 \leq m_2$, *then* $b_1 = b_2$ *and* $m_1' \leq m_2'$.

Lemma 20 *Assume* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b) = \lfloor m_2' \rfloor$. *If* $m_1 \leq m_2$, *then* $m_1' \leq m_2'$.

Proof Follows from Lemma 12 and property P34. □

```
// x implicitly initialized to undef
sp := alloc(0,8)
store(int, sp, 0, x)
...
y := load(int, sp, 0)
...
free(sp)
```

```
// R1 not initialized
sp := alloc(0,8)
store(int, sp, 0, R1)
...
R2 := load(int, sp, 0)
...
free(sp)
```

Fig. 6 Example of register allocation. *Left*: original code, *right*: transformed code. Variables `x` and `y` have been allocated to registers `R1` and `R2`, respectively

Lemma 21 *If* $m_1 \leq m_2$ *and* $\texttt{load}(\tau, m_1, b, i) = \lfloor v_1 \rfloor$, *then there exists a value* v_2 *such that* $\texttt{load}(\tau, m_2, b, i) = \lfloor v_2 \rfloor$ *and* $v_1 \leq v_2$.

Lemma 22 *If* $m_1 \leq m_2$ *and* $\texttt{store}(\tau, m_1, b, i, v_1) = \lfloor m_1' \rfloor$ *and* $v_1 \leq v_2$, *then there exists* m_2' *such that* $\texttt{store}(\tau, m_2, b, i, v_2) = \lfloor m_2' \rfloor$ *and* $m_1' \leq m_2'$.

5.4 Memory Injections

We now consider the most difficult memory transformation encountered in the Compcert development, during the translation from Clight to Cminor: the removal of some memory allocations performed by the Clight semantics and the coalescing of other memory allocations into sub-areas of a single block (see Fig. 2, left).

The pseudocode in Fig. 7 illustrates the effect of this transformation on the memory behavior of the program. Here, the transformation elected to "pull x out of memory", using a local variable x in the transformed program to hold the contents of the block pointed by x in the original program. It also merged the blocks pointed by y and z into a single block pointed by sp, with y corresponding to the sub-block at offsets [0, 8) and z to the sub-block at offsets [8, 10).

To relate the memory states in the original and transformed programs at any given point, we will again reuse the results on generic memory embeddings established in Section 5.1. However, unlike in Sections 5.2 and 5.3, we cannot work with a fixed embedding E, but need to build it incrementally during the proof of semantic preservation.

In the Compcert development, we use the following relation between values v_1 of the original Clight program and v_2 of the generated Cminor program, parametrized by an embedding E:

$$E \vdash \texttt{undef} \hookrightarrow v_2 \qquad E \vdash \texttt{int}(n) \hookrightarrow \texttt{int}(n) \qquad E \vdash \texttt{float}(n) \hookrightarrow \texttt{float}(n)$$

$$\frac{E(b_1) = \lfloor b_2, \delta \rfloor \qquad i_2 = i_1 + \delta}{E \vdash \texttt{ptr}(b_1, i_1) \hookrightarrow \texttt{ptr}(b_2, i_2)}$$

```
x := alloc(0, 4)                    sp := alloc(0, 10)
y := alloc(0, 8)
z := alloc(0, 2)
store(int, x, 0, 42)                x := 42
... load(int, x, 0) ...             ... x ...
store(double, y, 0, 3.14)           store(double, sp, 0, 3.14)
... load(short, z, 2) ...           ... load(short, sp, 8) ...
free(x)
free(y)
free(z)                             free(sp)
```

Fig. 7 Example of the Clight to Cminor translation. *Left*: original program, *right*: transformed program. Block x in the original program is pulled out of memory; its contents are stored in the local variable x in the transformed program. Blocks y and z become sub-blocks of sp, at offsets 0 and 8 respectively

In other words, `undef` Clight values can be refined by any Cminor value; integers and floating-point numbers must not change; and pointers are relocated as prescribed by the embedding E. Notice in particular that if $E(b) = \varepsilon$, there is no Cminor value v such that $E \vdash \mathtt{ptr}(b, i) \hookrightarrow v$. This means that the source Clight program is not allowed to manipulate a pointer value pointing to a memory block that we have decided to remove during the translation.

We assume that the $E \vdash v_1 \hookrightarrow v_2$ relation is compatible with the `convert` function: $E \vdash v_1 \hookrightarrow v_2$ implies $E \vdash \mathtt{convert}(v_1, \tau) \hookrightarrow \mathtt{convert}(v_2, \tau)$. (This clearly holds for the examples of `convert` functions given at the end of Section 2.)

We say that an embedding E injects a Clight memory state m_1 in a Cminor memory state m_2, and write $E \vdash m_1 \mapsto m_2$, if the following four conditions hold:

$$
\begin{aligned}
E \vdash m_1 \mapsto m_2 \quad \stackrel{\text{def}}{=} \quad & E \vdash m_1 \hookrightarrow m_2 && (1)\\
\wedge \ & \forall b_1, \ \ m_1 \# b_1 \ \Rightarrow \ E(b_1) = \varepsilon && (2)\\
\wedge \ & \forall b_1, b_2, \delta, \ \ E(b_1) = \lfloor b_2, \delta \rfloor \ \Rightarrow \ \neg(m_2 \# b_2) && (3)\\
\wedge \ & E \text{ is nonaliasing for } m_1 && (4)
\end{aligned}
$$

Condition (1) is the embedding of m_1 into m_2 in the sense of Section 5.1. Conditions (2) and (3) ensure that fresh blocks are not mapped, and that images of mapped blocks are not fresh. Condition (4) ensures that the embedding does not cause sub-blocks to overlap.

Using this definition, it is easy to show simulation results for the `load` and `store` operations performed by the original program.

Lemma 23 *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$ and $E(b_1) = \lfloor b_2, \delta \rfloor$, then there exists a value v_2 such that $\mathtt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$.*

Proof By (1) and definition of $E \vdash m_1 \hookrightarrow m_2$. □

Lemma 24 *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$ and $E(b_1) = \lfloor b_2, \delta \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$, then there exists m_2' such that $\mathtt{store}(\tau, m_2, b_2, i + \delta, v_2) = \lfloor m_2' \rfloor$ and $E \vdash m_1' \mapsto m_2'$.*

Proof Follows from Lemma 5. Conditions (2), (3) and (4) are preserved because of properties S16 and P33. □

Lemma 25 *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$ and $E(b_1) = \varepsilon$, then $E \vdash m_1' \mapsto m_2'$.*

Proof Follows from Lemma 3 and properties S16 and P33. □

In the Compcert development, given the algebra of values used (see Section 2), we can define the following variants `loadptr` and `storeptr` of `load` and `store` where the memory location being accessed is given as a pointer value:

```
loadptr(τ, m, a) =
  match a with ptr(b, i) ⇒ load(τ, m, b, i) | _ ⇒ ε
```

$$\begin{array}{l}\texttt{storeptr}(\tau, m, a, v) = \\ \quad \texttt{match}\ a\ \texttt{with}\ \texttt{ptr}(b, i) \Rightarrow \texttt{store}(\tau, m, b, i, v) \mid \_ \Rightarrow \varepsilon\end{array}$$

Lemmas 23 and 24 can then be restated in a more "punchy" way, taking advantage of the way $E \vdash v_1 \hookrightarrow v_2$ is defined over pointer values:

Lemma 26 *If $E \vdash m_1 \mapsto m_2$ and $\texttt{loadptr}(\tau, m_1, a_1) = \lfloor v_1 \rfloor$ and $E \vdash a_1 \hookrightarrow a_2$, then there exists a value v_2 such that $\texttt{loadptr}(\tau, m_2, a_2) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$.*

Lemma 27 *If $E \vdash m_1 \mapsto m_2$ and $\texttt{storeptr}(\tau, m_1, b_1, a_1, v_1) = \lfloor m_1' \rfloor$ and $E \vdash a_1 \hookrightarrow a_2$ and $E \vdash v_1 \hookrightarrow v_2$, then there exists m_2' such that $\texttt{storeptr}(\tau, m_2, b_2, a_2, v_2) = \lfloor m_2' \rfloor$ and $E \vdash m_1' \mapsto m_2'$.*

We now relate a sequence of deallocations performed in the original program with a single deallocation performed in the transformed program. (In the example of Fig. 7, this corresponds to the deallocations of `x`, `y` and `z` on one side and the deallocation of `sp` on the other side.) If l is a list of block references, we define the effect of deallocating these blocks as follows:

$$\begin{array}{l}\texttt{freelist}(m, l) = \\ \quad \texttt{match}\ l\ \texttt{with} \\ \quad\ \texttt{nil} \Rightarrow \lfloor m \rfloor \\ \quad \mid b :: l' \Rightarrow \texttt{match}\ \texttt{free}(m, b)\ \texttt{with}\ \varepsilon \Rightarrow \varepsilon \mid \lfloor m' \rfloor \Rightarrow \texttt{freelist}(m', l')\end{array}$$

Lemma 28 *Assume $\texttt{freelist}(m_1, l) = \lfloor m_1' \rfloor$ and $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$. Further assume that $E(b_1) = \lfloor b_2, \delta \rfloor \Rightarrow b_1 \in l$ for all b_1, δ; in other words, all blocks mapped to b_2 are in l and therefore are being deallocated from m_1. If $E \vdash m_1 \mapsto m_2$, then $E \vdash m_1' \mapsto m_2'$.*

Proof First, notice that for all $b_1 \in l$, $\neg(m_1' \models b_1)$, by S13 and P36. Part (1) of the expected result then follows from Lemmas 10 and 11. Parts (2) and (3) follow from P34. Part (4) follows by repeated application of Lemma 13. □

Symmetrically, we now consider a sequence of allocations performed by the original program and relate them with a single allocation performed by the transformed program. (In the example of Fig. 7, this corresponds to the allocations of `x`, `y` and `z` on one side and the allocation of `sp` on the other side.) A difficulty is that the current embedding E needs to be changed to map the blocks allocated by the original program; however, changing E should not invalidate the mappings for pre-existing blocks.

We say that an embedding E' is compatible with an earlier embedding E, and write $E \leq E'$, if, for all blocks b, either $E(b) = \varepsilon$ or $E'(b) = E(b)$. In other words, all blocks that are mapped by E remain mapped to the same target sub-block in E'. This relation is clearly reflexive and transitive. Moreover, it preserves injections between values:

Lemma 29 *If $E \vdash v_1 \hookrightarrow v_2$ and $E \leq E'$, then $E' \vdash v_1 \hookrightarrow v_2$.*

We first state and prove simulation lemmas for one allocation, performed either by the original program or by the transformed program. The latter case is straightforward:

Lemma 30 *If* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$, *then* $E \vdash m_1 \mapsto m_2'$.

Proof Follows from Lemma 7 and property P32. □

For an allocation performed by the original program, we distinguish two cases: either the new block is unmapped (Lemma 31), or it is mapped to a sub-block of the transformed program (Lemma 32).

Lemma 31 *Assume* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$. *Write* $E' = E\{b_1 \leftarrow \varepsilon\}$. *Then,* $E' \vdash m_1' \mapsto m_2$ *and* $E \leq E'$.

Proof By part (2) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property P31, it must be the case that $E(b_1) = \varepsilon$. It follows that $E' = E$, and therefore we have $E \leq E'$ and $E' \vdash m_1 \hookrightarrow m_2$. Applying Lemma 8, we obtain part (1) of the expected result $E' \vdash m_1' \mapsto m_2$. Part (2) follows from P32. Parts (3) and (4) are straightforward. □

Lemma 32 *Assume* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $m_2 \models b_2$ *and* $\mathcal{L}(m_2, b_2) \leq l + \delta$ *and* $h + \delta \leq \mathcal{H}(m_2, b_2)$ *and* `max_alignment` *divides* δ. *Further assume that for all blocks* b' *and offsets* δ',

$$E(b') = \lfloor b_2, \delta' \rfloor \;\Rightarrow\; \mathcal{H}(m_1, b') + \delta' \leq l + \delta \;\vee\; h + \delta \leq \mathcal{L}(m_1, b') + \delta' \qquad (*)$$

Write $E' = E\{b_1 \leftarrow \lfloor b_2, \delta \rfloor\}$. *If* $E \vdash m_1 \mapsto m_2$, *then* $E' \vdash m_1' \mapsto m_2$ *and* $E \leq E'$.

Proof By part (2) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property P31, it must be the case that $E(b_1) = \varepsilon$. It follows that $E \leq E'$.

We first show that $E' \vdash m_1 \hookrightarrow m_2$. Assume $E'(b) = \lfloor b', \delta' \rfloor$ and $\texttt{load}(\tau, m_1, b, i) = \lfloor v \rfloor$. It must be the case that $b \neq b_1$, since b_1 is not valid in m_1. Therefore, $E(b) = \lfloor b', \delta' \rfloor$ and the result follows from part (1) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and from Lemma 29.

Using Lemma 9, we obtain part (1) of the expected result $E' \vdash m_1' \mapsto m_2$. Part (2) follows from P32. Part (3) follows from the fact that b_2 is not fresh (property P30). Finally, part (4) follows from hypothesis (*) and property S15. □

We now define the `alloclist` function, which, given a list L of (low, high) bounds, allocates the corresponding blocks and returns both the list B of their references and the final memory state. In the example of Fig. 7, the allocation of `x`, `y` and `z` corresponds to an invocation of `alloclist` with the list $L = (0, 4); (0, 8); (0, 2)$.

```
alloclist(m, L) =
  match L with
   nil ⇒ ⌊nil, m⌋
  | (l, h) :: L' ⇒
     match alloc(m, l, h) with
      ε ⇒ ε
```

$$
\begin{array}{l}
\mid \lfloor b, m' \rfloor \Rightarrow \\
\quad \texttt{match alloclist}(m', L') \texttt{ with} \\
\quad\ \varepsilon \Rightarrow \varepsilon \\
\quad \mid \lfloor B, m'' \rfloor \Rightarrow \lfloor b :: B, m'' \rfloor
\end{array}
$$

Along with the list $L = (l_1, h_1) \ldots (l_n, h_n)$ of allocation requests to be performed in the original program, we assume given the bounds (l, h) of a block to be allocated in the transformed program, and a list $P = p_1, \ldots, p_n$ of elements of type `option` $\mathbb{Z}$, indicating how these allocated blocks should be mapped in the transformed program. If $p_i = \varepsilon$, the i-th block is unmapped, but if $p_i = \lfloor \delta_i \rfloor$, it should be mapped at offset δ_i. In the example of Fig. 7, we have $l = 0$, $h = 10$, $p_1 = \varepsilon$, $p_2 = \lfloor 0 \rfloor$, and $p_3 = \lfloor 8 \rfloor$.

We say that the quadruple (L, P, l, h) is well-formed if the following conditions hold:

1. L and P have the same length.
2. If $p_i = \lfloor \delta_i \rfloor$, then $l \leq l_i + \delta_i$ and $h_i + \delta_i \leq h$ and `max_alignment` divides δ_i (the image of the i-th block is within bounds and aligned).
3. If $p_i = \lfloor \delta_i \rfloor$ and $p_j = \lfloor \delta_j \rfloor$ and $i \neq j$, then $h_i + \delta_i \leq l_j + \delta_j$ or $h_j + \delta_j \leq l_i + \delta_i$ (blocks are mapped to disjoint sub-blocks).

Lemma 33 *Assume that (L, P, l, h) is well-formed. Assume* `alloclist`$(m_1, L) = \lfloor B, m_1' \rfloor$ *and* `alloc`$(m_2, l, h) = \lfloor b, m_2' \rfloor$. *If $E \vdash m_1 \mapsto m_2$, there exists an embedding E' such that $E' \vdash m_1' \mapsto m_2'$ and $E \leq E'$. Moreover, writing b_i for the i-th element of B and p_i for the i-th element of P, we have $E'(b_i) = \varepsilon$ if $p_i = \varepsilon$, and $E'(b_i) = \lfloor b, \delta_i \rfloor$ if $p_i = \lfloor \delta_i \rfloor$.*

Proof By Lemma 30, we have $E \vdash m_1 \mapsto m_2'$. We then show the expected result by induction over the length of the lists L and P, using an additional induction hypothesis: for all b', δ, i, if $E(b') = \lfloor b, \delta \rfloor$ and $p_i = \lfloor \delta' \rfloor$, then $h_i + \delta_i \leq \mathcal{L}(m_1, b') + \delta$ or $\mathcal{H}(m_1, b') + \delta \leq l_i + \delta_i$. In other words, the images of mapped blocks that remain to be allocated are disjoint from the images of the mapped blocks that have already been allocated. The proof uses Lemmas 31 and 32. We finish by proving the additional induction hypothesis for the initial state, which is easy since the initial embedding E does not map any block to a sub-block of the fresh block b. □

6 Mechanical Verification

We now briefly comment on the Coq mechanization of the results presented in this article, which can be consulted on-line at http://gallium.inria.fr/~xleroy/memory-model/. The Coq development is very close to what is presented here. Indeed, almost all specifications and statements of theorems given in this article were transcribed directly from the Coq development. The only exception is the definition of well-formed multiple allocation requests at the end of Section 5.4, which is presented as inductive predicates in the Coq development, such predicates being easier to reason upon inductively than definitions involving i-th elements of lists. Also, the Coq development proves additional lemmas not shown in this paper: 8 derived properties in the style of properties D19-D22, used to shorten the proofs of Section 5, and 16 properties of auxiliary functions occurring in the concrete

implementation of the model, used in the proofs of Section 4, especially that of Lemma 1.

The mechanized development uses the Coq module system [6] to clearly separate specifications from implementations. The specification of the abstract memory model from Section 3, as well as the properties of the concrete memory model from Section 4, are given as module signatures. The concrete implementation of the model is a structure that satisfies these two signatures. The derived properties of Sections 3 and 4, as well as the memory transformations of Section 5, are presented as functors, i.e., modules parametrized by any implementation of the abstract signature or concrete signature, respectively. This use of the Coq module system ensures that the results we proved, especially those of Section 5, do not depend on accidental features of our concrete implementation, but only on the properties stated earlier.

The Coq module system was effective at enforcing this kind of abstraction. However, we hit one of its limitations: no constructs are provided to extend a posteriori a module signature (interface) with additional declarations and logical properties. The Standard ML and Objective Caml module systems support such extensions through the `open` and `include` constructs, respectively. By lack of such constructs in Coq, the signature of the abstract memory model must be manually duplicated in the signature of the concrete memory model, and later changes to the abstract signature must be manually propagated to the concrete signature. For our development, this limitation was a minor annoyance, but it is likely to cause serious problems for developments that involve a large number of refinement steps.

The Coq development represents approximately 1070 non-blank lines of specifications and statements of theorems, and 970 non-blank lines of proof scripts. Most of the proofs are conducted manually, since Coq does not provide much support for proof automation. However, our proofs intensively use the `omega` tactic, a decision procedure for Presburger arithmetic that automates reasoning about linear equalities and inequalities. The `eauto` (Prolog-style resolution) and `congruence` (equational reasoning via the congruence closure algorithm) tactics were also occasionally useful, but the `tauto` and `firstorder` tactics (propositional and first-order automatic reasoning, respectively) were either too weak (`tauto`) or too inefficient (`firstorder`) to be useful.

As pointed out by one of the reviewers, our formalization is conducted mostly in first-order logic: functions are used as data in Sections 4 and 5, but only to implement finite maps, which admit a simple, first-order axiomatization. A legitimate question to ask, therefore, is whether our proofs could be entirely automated using a modern theorem prover for first-order logic. We experimented with this approach using three automatic theorem provers: Ergo [7], Simplify [10] and Z3 [9]. The Why platform for program proof [12] was used to administer the tests and to translate automatically between the input syntaxes of the provers. Most of the Coq development was translated to Why's input syntax. (The only part we did not translate is the concrete implementation of the memory model, because it uses recursive functions that are difficult to express in this syntax.) Each derived property and lemma was given to the three provers as a goal, with a time limit of 5 min of CPU time.[4]

[4]The test was run on a 2.4 GHz Intel Core 2 Duo processor, with 2 Gb of RAM, running the MacOS 10.4 operating system. The versions of the provers used are: Ergo 0.7.2, compiled with OCaml version 3.10.1; Simplify 1.5.5; Z3 1.1, running under CrossOver Mac.

Table 1 Experiments with three automated theorem provers

	Ergo	Simplify	Z3	At least one
Derived properties from Sections 3 and 4	15/15	15/15	15/15	15/15
Generic memory embeddings (Section 5.1)	7/12	1/12	6/12	9/12
Memory extensions (Section 5.2)	0/7	1/7	5/7	5/7
Refinement of stored values (Section 5.3)	2/8	3/8	6/8	6/8
Memory injections (Section 5.4)	4/8	4/8	7/8	7/8
Total	28/50	24/50	39/50	42/50

Table 1 summarizes the results of this experiment. A total of 50 goals were given to the three provers: the derived properties D19-D22 and D29 from Sections 3 and 4, all lemmas from Section 5, and some auxiliary lemmas present in the Coq development. Of these 50 goals, 42 were proved by at least one of the three provers. The 8 goals that all three provers fail to establish within the 5-min time limit correspond to Lemmas 5, 6, 12, 15, 17, 19, 22, and 31 from Section 5. These preliminary results are encouraging: while interactive proof remains necessary for some of the most difficult theorems, integration of first-order theorem proving within a proof assistant has great potential to significantly shorten our proofs.

7 Related Work

Giving semantics to imperative languages and reasoning over pointer programs has been the subject of much work since the late 1960's. Reynolds [23] and Tennent and Ghica [26] review some of the early work in this area. In the following, we mostly focus on semantics and verifications that have been mechanized.

For the purpose of this discussion, memory models can be roughly classified as either "high level", where the model itself provides some guarantees of separation, enforcement of memory bounds, etc., or "low-level", where the memory is modeled essentially as an array of bytes and such guarantees must be enforced through additional logical assertions.

A paradigmatic example of high-level modeling is the Burstall-Bornat encoding of records (`struct`), where each field is viewed as a distinct memory store mapping addresses to contents [4, 5]. Such a representation captures the fact that distinct fields of a `struct` value are separated: it becomes obvious that assigning to one field though a pointer (`p->f = x` in C) leaves unchanged the values of any other field. In turn, this separation guarantee greatly facilitates reasoning over programs that manipulate linked data structures, as demonstrated by Mehta and Nipkow [19] and the Caduceus program prover of Filliâtre and Marché [11]. However, this representation makes it difficult to account for other features of the C language, such as `union` types and some casts between pointer types.

Examples of low-level modeling of memory include Norrish's HOL semantics for C [21] and the work of Tuch et al. [27]. There, a memory state is essentially a mapping from addresses to bytes, and allocation, loads and stores are axiomatized in these terms. The axioms can either leave unspecified all behaviors undefined in the C standard, or specify additional behaviors arising from popular violations

of the C standard such as casts between incompatible pointer types. Reasoning about programs and program transformations is more difficult than with a high-level memory model; Tuch et al. [27] use separation logic to alleviate these difficulties.

The memory model presented in this article falls half-way between high-level models and low-level models. It guarantees several useful properties: separation between blocks obtained by distinct calls to `alloc`, enforcement of bounds during memory accesses, and the fact that loads overlapping a prior store operation predictably return the `undef` value. These properties play a crucial role in verifying the program transformations presented in Section 5. In particular, a lower-level memory model where a load overlapping a previous store could return an unspecified value would invalidate the simulation lemmas for memory injections (Section 5.4). On the other hand, the model offers no separation guarantees over accesses performed within the same memory block. The natural encoding of a `struct` value as a single memory block does not, by itself, validate the Burstall-Bornat separation properties; additional reasoning over field offsets is required.

Separation logic, introduced by O'Hearn et al. [22, 24], and the related spatial logic of Jia and Walker [14], provide an elegant way to reason over memory separation properties of pointer programs. Central to these approaches is the separating conjunction $P * Q$, which guarantees that the logical assertions P and Q talk about disjoint areas of the memory state. Examples of use of separation logic include correctness proofs for memory allocators and garbage collectors [17, 18]. It is possible, but not very useful, to define a separation logic on top of our memory model, where in a separating conjunction $P * Q$, every memory block is wholly owned by either P or Q but not both. Appel and Blazy [1] develop a finer-grained separation logic for the Cminor intermediate language of Compcert where disjoint parts of a given block can be separated.

While intended for sequential programs, the memory model described in this paper can also be used to describe concurrent executions in a strongly-consistent shared memory context, where the memory effect of a concurrent program is equivalent to an interleaving of the load and store operations performed by each of its threads. Modern multiprocessor systems implement weakly-consistent forms of shared memory, where the execution of a concurrent program cannot be described as such interleavings of atomic load and store operations. The computer architecture community has developed sophisticated hardware memory models to reason about weakly-consistent memory. For instance, Shen et al. [25] use a term rewriting system to define a memory model that decomposes load and store operations into finer-grained operations. This model formalizes the notions of data replication and instruction reordering. It aims at defining the legal behaviors of a distributed shared-memory system that relies on execution trace of memory accesses. Another example of architecture-centric memory model is that of Yang et al. [28].

Going back to memory models for programming languages, features of architectural models for weakly-consistent shared memory also appear in specifications of programming languages that support shared-memory concurrency. A famous example is Java, whose specification of its memory model has gone through several iterations. Manson et al. [16] describe and formalize the latest version of the Java memory model. Reasoning over concurrent, lock-free programs in Java or any other shared-memory, weakly-consistent concurrency model remains challenging. Reasoning over transformations of such programs is an open problem.

Software written in C, especially systems code, often makes assumptions about the layout of data in memory and the semantics of memory accesses that are left unspecified by the C standard. Recent work by Nita et al. [20] develops a formal framework to characterize these violations of the C standard and to automatically detect the portability issues they raise. Central to their approach is the notion of a *platform*, which is an abstract description of the assumptions that non-portable code makes about concrete data representations. Some aspects of their notion of platform are captured by our memory model, via the size and alignment functions and the type compatibility relation from Section 2. However, our model does not account for many other aspects of platforms, such as the layout and padding algorithm for `struct` types.

8 Conclusions

We have presented and formalized a software memory model at a level of abstraction that is intermediate between the high-level view of memory that underlies the C standard and the low-level view of memory that is implemented in hardware. This memory model is adequate for giving semantics and reasoning over intermediate languages typically found in compilers. In particular, the main features of our model (separation between blocks, bounds checking, and the `undef` value resulting from ill-defined loads) played a crucial role in proving semantics preservation for the Compcert verified compiler.

This model can also be used to give a concrete semantics for the C language that specifies the behavior of a few popular violations of the C standard, such as arbitrary casts between pointers, as well as pointer arithmetic within `struct` types. However, many other violations commonly used in systems code or run-time systems for programming languages (for instance, copying arrays of characters 4 or 8 elements as a time using integer or floating-point loads and stores) cannot be accounted for in our model. We have considered several variants of our model that could give meaning to these idioms, but have not yet found one that would still validate all simulation properties of Section 5. This remains an important direction for future work, since such a model would be useful not only to reason over systems C code, but also to prove that the semantics of such code is preserved during compilation by the Compcert compiler.

Another direction for future work is to construct and prove correct refinements from a high-level model such as the Burstall-Bornat model used in Caduceus [11] to our model, and from our model to a low-level, hardware-oriented memory model. Such refinements would strengthen the proof of semantic preservation of the Compcert compiler, which currently uses a single memory model for the source and target languages.

Acknowledgements Our early explorations of memory models for the Compcert project were conducted in collaboration with Benjamin Grégoire and François Armand, and benefited from discussions with Catherine Dubois and Pierre Letouzey. Sylvain Conchon, Jean-Christophe Filliâtre and Benjamin Monate helped us experiment with automatic theorem provers. We thank the anonymous reviewers as well as Nikolay Kosmatov for their careful reading of this article and their helpful suggestions for improvements.

References

1. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007, Lecture Notes in Computer Science, vol. 4732, pp. 5–21. Springer, New York (2007)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer, New York (2004)
3. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods, Lecture Notes in Computer Science, vol. 4085, pp. 460–475. Springer, New York (2006)
4. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC '00: Proc. Int. Conf. on Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 1837, pp. 102–126. Springer, New York (2000)
5. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Mach. Intell. **7**, 23–50 (1972)
6. Chrząszcz, J.: Modules in type theory with generative definitions. Ph.D. thesis, Warsaw University and University of Paris-Sud (2004)
7. Conchon, S., Contejean, E., Kanig, J.: The Ergo automatic theorem prover. Software and documentation available at http://ergo.lri.fr/ (2005–2008)
8. Coq development team: The Coq proof assistant. Software and documentation available at http://coq.inria.fr/ (1989–2008)
9. De Moura, L., Bjørner, N., et al.: Z3: an efficient SMT solver. Software and documentation available at http://research.microsoft.com/projects/z3 (2006–2008)
10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
11. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: 6th Int. Conf. on Formal Engineering Methods, ICFEM 2004, Lecture Notes in Computer Science, vol. 3308, pp. 15–29. Springer, New York (2004)
12. Filliâtre, J.C., Marché, C., Moy, Y., Hubert, T.: The Why software verification platform. Software and documentation available at Software and documentation available at http://why.lri.fr/ (2004–2008)
13. ISO: International Standard ISO/IEC 9899:1999, Programming languages – C. ISO, Geneva (1999)
14. Jia, L., Walker, D.: ILC: a foundation for automated reasoning about pointer programs. In: Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Lecture Notes in Computer Science, vol. 3924, pp. 131–145. Springer, New York (2006)
15. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd Symposium Principles of Programming Languages, pp. 42–54. ACM, New York (2006)
16. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: 32nd Symposium Principles of Programming Languages, pp. 378–391. ACM, New York (2005)
17. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an operating system using separation logic. In: Formal Methods and Software Engineering, 8th Int. Conf. ICFEM 2006, Lecture Notes in Computer Science, vol. 4260, pp. 400–419. Springer, New York (2006)
18. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Programming Language Design and Implementation 2007, pp. 468–479. ACM, New York (2007)
19. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2), 200–227 (2005)
20. Nita, M., Grossman, D., Chambers, C.: A theory of platform-dependent low-level software. In: 35th Symposium Principles of Programming Languages. ACM, New York (2008)
21. Norrish, M.: C formalized in HOL. Technical report UCAM-CL-TR-453. Ph.D. thesis, University of Cambridge (1998)
22. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic, 15th Int. Workshop, CSL 2001, Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer, New York (2001)

23. Reynolds, J.C.: Intuitionistic reasoning about shared data structures. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millenial Perspectives in Computer Science, pp. 303–321. Palgrave, Hampshire (2000)
24. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
25. Shen, X., Arvind, R.L.: Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In: ISCA '99: Proc. Int. Symp. on Computer Architecture, pp. 150–161. IEEE Computer Society, Los Alamitos (1999)
26. Tennent, R.D., Ghica, D.R.: Abstract models of storage. Higher-Order and Symbolic Computation **13**(1–2), 119–129 (2000)
27. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: 34th Symposium Principles of Programming Languages, pp. 97–108. ACM, New York (2007)
28. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurr. Comput.: Practice and Experience **17**(5–6), 465–487 (2005)

Separation Logic for Small-Step Cminor

Andrew W. Appel[1,⋆] and Sandrine Blazy[2,⋆]

[1] Princeton University
[2] ENSIIE

Abstract. Cminor is a mid-level imperative programming language; there are proved-correct optimizing compilers from C to Cminor and from Cminor to machine language. We have redesigned Cminor so that it is suitable for Hoare Logic reasoning and we have designed a Separation Logic for Cminor. In this paper, we give a small-step semantics (instead of the big-step of the proved-correct compiler) that is motivated by the need to support future concurrent extensions. We detail a machine-checked proof of soundness of our Separation Logic. This is the first large-scale machine-checked proof of a Separation Logic w.r.t. a small-step semantics. The work presented in this paper has been carried out in the Coq proof assistant. It is a first step towards an environment in which concurrent Cminor programs can be verified using Separation Logic and also compiled by a proved-correct compiler with formal end-to-end correctness guarantees.

1 Introduction

The future of program verification is to connect machine-verified source programs to machine-verified compilers, and run the object code on machine-verified hardware. To connect the verifications end to end, the source language should be specified as a structural operational semantics (SOS) represented in a logical framework; the target architecture can also be specified that way. Proofs of source code can be done in the logical framework, or by other tools whose soundness is proved w.r.t. the SOS specification; these may be in safety proofs via type-checking, correctness proofs via Hoare Logic, or (in source languages designed for the purpose) correctness proofs by a more expressive proof theory. The compiler—if it is an optimizing compiler—will be a stack of phases, each with a well specified SOS of its own. There will be proofs of (partial) correctness of each compiler phase, or witness-driven recognizers for correct compilations, w.r.t. the SOS's that are inputs and outputs to the phases.

Machine-verified hardware/compiler/application stacks have been built before. Moore described a verified compiler for a "high-level assembly language" [13]. Leinenbach *et al.* [11] have built and proved a compiler for *C0*, a small C-like language, as part of a project to build machine-checked correctness proofs

⋆ Appel supported in part by NSF Grants CCF-0540914 and CNS-0627650. This work was done, in part, while both authors were on sabbatical at INRIA.

K. Schneider and J. Brandt (Eds.): TPHOLs 2007, LNCS 4732, pp. 5–21, 2007.

of source programs, Hoare Logic, compiler, micro-kernel, and RISC processor. These are both simple one- or two-pass nonoptimizing compilers.

Leroy [12] has built and proved correct in Coq [1] a compiler called *CompCert* from a high-level intermediate language *Cminor* to assembly language for the Power PC architecture. This compiler has 4 intermediate languages, allowing optimizations at several natural levels of abstraction. Blazy *et al.* have built and proved correct a translator from a subset of C to Cminor [5]. Another compiler phase on top (not yet implemented) will then yield a proved-correct compiler from C to machine language. We should therefore reevaluate the conventional wisdom that an entire practical optimizing compiler cannot be proved correct.

A software system can have components written in different languages, and we would like end-to-end correctness proofs of the whole system. For this, we propose a new variant of Cminor as a machine-independent intermediate language to serve as a common denominator between high-level languages. Our new Cminor has a usable Hoare Logic, so that correctness proofs for some components can be done directly at the level of Cminor.

Cminor has a "calculus-like" view of local variables and procedures (*i.e.* local variables are bound in an environment), while Leinenbach's C0 has a "storage-allocation" view (*i.e.* local variables are stored in the stack frame). The calculus-like view will lead to easier reasoning about program transformations and easier use of Cminor as a target language, and fits naturally with a multi-pass optimizing compiler such as CompCert; the storage-allocation view suits the one-pass nonoptimizing C0 compiler and can accommodate in-line assembly code.

Cminor is a promising candidate as a common intermediate language for end-to-end correctness proofs. But we have many demands on our new variant of Cminor, only the first three of which are satisfied by Leroy's Cminor.

- Cminor has an operational semantics represented in a logical framework.
- There is a proved-correct compiler from Cminor to machine language.
- Cminor is usable as the high-level target language of a C compiler.
  - Our semantics is a *small-step* semantics, to support reasoning about input/output, concurrency, and nontermination.
  - Cminor is machine-independent over machines in the "standard model" (*i.e.* 32- or 64-bit single-address-space byte-addressable multiprocessors).
  - Cminor can be used as a mid-level target language of an ML compiler [8], or of an OO-language compiler, so that we can integrate correctness proofs of ML or OO programs with the proofs of their run-time systems and libraries.
  - As we show in this paper, Cminor supports an axiomatic Hoare Logic (in fact, Separation Logic), proved sound with respect to the small-step semantics, for reasoning about low-level (C-like) programs.
  - In future work, we plan to extend Cminor to be concurrent in the "standard model" of thread-based preemptive lock-synchronized weakly consistent shared-memory programming. The sequential soundness proofs we present here should be reusable in a concurrent setting, as we will explain.

Leroy's original Cminor had several Power-PC dependencies, is slightly clumsy to use as the target of an ML compiler, and is a bit clumsy to use in Hoare-style

reasoning. But most important, Leroy's semantics is a big-step semantics that can be used only to reason about terminating sequential programs. We have redesigned Cminor's syntax and semantics to achieve all of these goals. That part of the redesign to achieve target-machine portability was done by Leroy himself. Our redesign to ease its use as an ML back end and for Hoare Logic reasoning was fairly simple. Henceforth in this paper, Cminor will refer to the new version of the Cminor language.

The main contributions of this paper are a small-step semantics suitable for compilation and for Hoare Logic; and the first machine-checked proof of soundness of a sequential Hoare Logic (Separation Logic) w.r.t. a small-step semantics. Schirmer [17] has a machine-checked *big-step* Hoare-Logic soundness proof for a control flow much like ours, extended by Klein *et al.* [10] to a C-like memory model. Ni and Shao [14] have a machine-checked proof of soundness of a Hoare-like logic w.r.t. a small-step semantics, but for an assembly language and for much simpler assertions than ours.

2 Big-Step Expression Semantics

The C standard [2] describes a memory model that is byte- and word-addressable (yet portable to big-endian and little-endian machines) with a nontrivial semantics for uninitialized variables. Blazy and Leroy formalized this model [6] for the semantics of Cminor. In C, pointer arithmetic within any malloc'ed block is defined, but pointer arithmetic between different blocks is undefined; Cminor therefore has non-null pointer values comprising an abstract block-number and an int offset. A `NULL` pointer is represented by the integer value 0. Pointer arithmetic between blocks, and reading uninitialized variables, are undefined but not illegal: expressions in Cminor can evaluate to *undefined* (Vundef) without getting stuck.

Each memory load or store is to a non-null pointer value with a "chunk" descriptor *ch* specifying number of bytes, signed or unsigned, int or float. Storing as 32-bit-int then loading as 8-bit-signed-byte leads to an undefined value. Load and store operations on memory, $m \vdash v_1 \overset{ch}{\mapsto} v_2$ and $m' = m[v_1 \overset{ch}{:=} v_2]$, are partial functions that yield results only if reading (resp., writing) a chunk of type *ch* at address v_1 is legal. We write $m \vdash v_1 \overset{ch}{\mapsto} v$ to mean that the result of loading from memory m at address v_1 a chunk-type *ch* is the value v.

The *values* of Cminor are *undefined* (Vundef), integers, pointers, and floats. The int type is an abstract data-type of 32-bit modular arithmetic. The expressions of Cminor are literals, variables, primitive operators applied to arguments, and memory loads.

There are 33 primitive operation symbols *op*; two of these are for accessing global names and local stack-blocks, and the rest is for integer and floating-point arithmetic and comparisons. Among these operation symbols are casts. Cminor casts correspond to all portable C casts. Cminor has an infinite supply ident of variable and function identifiers *id*. As in C, there are two namespaces—each *id*

can be interpreted in a local scope (using $\mathsf{Evar}\,(id)$) or in a global scope (using Eop with the operation symbol for accessing global names).

$$\begin{aligned} i &: \mathsf{int} ::= [0, 2^{32}) \\ v &: \mathsf{val} ::= \mathsf{Vundef} \mid \mathsf{Vint}\,(i) \mid \mathsf{Vptr}\,(b,i) \mid \mathsf{Vfloat}\,(f) \\ e &: \mathsf{expr} ::= \mathsf{Eval}\,(v) \mid \mathsf{Evar}\,(id) \mid \mathsf{Eop}\,(op, el) \mid \mathsf{Eload}\,(ch, e) \\ el &: \mathsf{exprlist} ::= \mathsf{Enil} \mid \mathsf{Econs}\,(e, el) \end{aligned}$$

Expression Evaluation. In original Cminor, expression evaluation is expressed by an inductive big-step relation. Big-step statement execution is problematic for concurrency, but big-step *expression* evaluation is fine even for concurrent programs, since we will use the separation logic to prove noninterference.

Evaluation is deterministic. Leroy chose to represent evaluation as a relation because Coq had better support for proof induction over relations than over function definitions. We have chosen to represent evaluation as a partial function; this makes some proofs easier in some ways: $f(x) = f(x)$ is simpler than $f\,x\,y \Rightarrow f\,x\,z \Rightarrow y = z$. Before Coq's new functional induction tactic was available, we developed special-purpose tactics to enable these proofs. Although we specify expression evaluation as a function in Coq, we present evaluation as a judgment relation in Fig. 1. Our evaluation function is (proved) equivalent to the inductively defined judgment $\Psi; (sp; \rho; \phi; m) \vdash e \Downarrow v$ where:

- Ψ is the "program," consisting of a global environment ($\mathsf{ident} \to \mathsf{option\,block}$) mapping identifiers to function-pointers and other global constants, and a global mapping ($\mathsf{block} \to \mathsf{option\,function}$) that maps certain ("text-segment") addresses to function definitions.
- sp : block. The "stack pointer" giving the address and size of the memory block for stack-allocated local data in the current activation record.
- ρ : env. The local environment, a finite mapping from identifiers to values.
- ϕ : $\mathsf{footprint}$. It represents the memory used by the evaluation of an expression (or a statement). It is a mapping from memory addresses to permissions. Leroy's Cminor has no footprints.
- m : mem. The memory, a finite mapping from blocks to block contents [6]. Each block represents the result of a C `malloc`, or a stack frame, a global static variable, or a function code-pointer. A block content consists of the dimensions of the block (low and high bounds) plus a mapping from byte offsets to byte-sized memory cells.
- e : expr. The expression being evaluated.
- v : val. The value of the expression.

Loads outside the footprint will cause expression evaluation to get stuck. Since the footprint may have different permissions for loads than for stores to some addresses, we write $\phi \vdash \mathsf{load}_{ch}\,v$ (or $\phi \vdash \mathsf{store}_{ch}\,v$) to mean that all the addresses from v to $v + |ch| - 1$ are readable (or writable).

To model the possibility of exclusive read/write access or shared read-only access, we write $\phi_0 \oplus \phi_1 = \phi$ for the "disjoint" sum of two footprints, where $\oplus$

$$\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Eval}\,(v) \Downarrow v \qquad \frac{x \in \mathrm{dom}\,\rho}{\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Evar}\,(x) \Downarrow \rho(x)}$$

$$\frac{\Psi; (sp; \rho; \phi; m) \vdash el \Downarrow vl \qquad \Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v}{\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Eop}\,(op, el) \Downarrow v}$$

$$\frac{\Psi; (sp; \rho; \phi; m) \vdash e_1 \Downarrow v_1 \qquad \phi \vdash \mathsf{load}_{ch}\, v_1 \qquad m \vdash v_1 \overset{ch}{\mapsto} v}{\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Eload}\,(ch, e_1) \Downarrow v}$$

Fig. 1. Expression evaluation rules

is an associative and commutative operator with several properties such as $\phi_0 \vdash \mathsf{store}_{ch}\, v \;\Rightarrow\; \phi_1 \not\vdash \mathsf{load}_{ch}\, v$, $\phi_0 \vdash \mathsf{load}_{ch}\, v \;\Rightarrow \phi \vdash \mathsf{load}_{ch}\, v$ and $\phi_0 \vdash \mathsf{store}_{ch}\, v \;\Rightarrow \phi \vdash \mathsf{store}_{ch}\, v$. One can think of ϕ as a set of fractional permissions [7], with 0 meaning no permission, $0 < x < 1$ permitting read, and 1 giving read/write permission. A store permission can be split into two or more load permissions, which can be reconstituted to obtain a store permission. Instead of fractions, we use a more general and powerful model of sharable permissions similar to one described by Parkinson [16, Ch. 5].

Most previous models of Separation Logic (*e.g.*, Ishtiaq and O'Hearn [9]) represent heaps as partial functions that can be combined with an operator like $\oplus$. Of course, a partial function can be represented as a pair of a domain set and a total function. Similarly, we represent heaps as a footprint plus a Cminor memory; this does not add any particular difficulty to the soundness proofs for our Separation Logic.

To perform arithmetic and other operations, in the third rule of Fig. 1, the judgment $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ takes an operator op applied to a list of values vl and (if vl contains appropriate values) produces some value v. Operators that access global names and local stack-blocks make use of Ψ and sp respectively to return the address of a global name or a local stack-block address.

States. We shall bundle together $(sp; \rho; \phi; m)$ and call it the *state*, written as σ. We write $\Psi; \sigma \vdash e \Downarrow v$ to mean $\Psi; (sp_\sigma; \rho_\sigma; \phi_\sigma; m_\sigma) \vdash e \Downarrow v$.

Notation. We write $\sigma[:= \rho']$ to mean the state σ with its environment component ρ replaced by ρ', and so on (*e.g.* see rules 2 and 3 of Fig. 2 in Section 4).

Fact. $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ and $m \vdash v_1 \overset{ch}{\mapsto} v$ are both deterministic relations, *i.e.* functions.

Lemma 1. *$\Psi; \sigma \vdash e \Downarrow v$ is a deterministic relation. (Trivial by inspection.)*

Lemma 2. *For any value v, there is an expression e such that $\forall \sigma.\ (\Psi; \sigma \vdash e \Downarrow v)$.*

Proof. Obvious; e is simply $\mathsf{Eval}\, v$*. But it is important nonetheless: reasoning about programs by rewriting and by Hoare Logic often requires this property, and it was absent from Leroy's Cminor for* Vundef *and* Vptr *values.* ■

An expression may fetch from several different memory locations, or from the same location several times. Because $\Downarrow$ is deterministic, we cannot model a situation where the memory is updated by another thread after the first fetch and before the second. But we want a semantics that describes real executions on real machines. The solution is to evaluate expressions in a setting where we can guarantee *noninterference*. We will do this (in our extension to Concurrent Cminor) by guaranteeing that the footprints ϕ of different threads are disjoint.

Erased Expression Evaluation. The Cminor compiler (CompCert) is proved correct w.r.t. an operational semantics that does not use footprints. Any program that successfully evaluates with footprints will also evaluate ignoring footprints. Thus, for sequential programs where we do not need noninterference, it is sound to prove properties in a footprint semantics and compile in an erased semantics. We formalize and prove this in the full technical report [4].

3 Small-Step Statement Semantics

The statements of sequential Cminor are:

$$\begin{aligned} s : \mathsf{stmt} \quad ::= {} & x := e \mid [e_1]_{ch} := e_2 \mid \mathsf{loop}\, s \mid \mathsf{block}\, s \mid \mathsf{exit}\, n \\ & \mid \mathsf{call}\, xl\, e\, el \mid \mathsf{return}\, el \mid s_1; s_2 \mid \mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \mid \mathsf{skip}. \end{aligned}$$

The assignment $x := e$ puts the value of e into the local variable x. The store $[e_1]_{ch} := e_2$ puts (the value of) e_2 into the memory-chunk ch at address given by (the value of) e_1. (Local variables are not addressable; global variables and heap locations are memory addresses.) To model exits from nested loops, $\mathsf{block}\, s$ runs s, which should not terminate normally but which should $\mathsf{exit}\, n$ from the $(n+1)^{th}$ enclosing block, and $\mathsf{loop}\, s$ repeats s infinitely or until it returns or exits. $\mathsf{call}\, xl\, e\, el$ calls function e with parameters (by value) el and results returned back into the variables xl. $\mathsf{return}\, el$ evaluates and returns a sequence of results, $(s_1; s_2)$ executes s_1 followed by s_2 (unless s_1 returns or exits), and the statements if and skip are as the reader might expect.

Combined with infinite loops and if statements, blocks and exits suffice to express efficiently all reducible control-flow graphs, notably those arising from C loops. The C statements break and $\mathsf{continue}$ are translated as appropriate exit statements. Blazy *et al.* [5] detail the translation of these C statements into Cminor.

Function Definitions. A program Ψ comprises two mappings: a mapping from function names to memory blocks (*i.e.*, abstract addresses), and a mapping from memory blocks to function definitions. Each function definition may be written as $f = (xl, yl, n, s)$, where $\mathsf{params}(f) = xl$ is a list of formal parameters, $\mathsf{locals}(f) = yl$ is a list of local variables, $\mathsf{stackspace}(f) = n$ is the size of the local stack-block to which sp points, and the statement $\mathsf{body}(f) = s$ is the function body.

Operational Semantics. Our small-step semantics for statements is based on continuations, mainly to allow a uniform representation of statement execution that facilitates the design of lemmas. Such a semantics also avoids all search

$$\Psi \vdash (\sigma,\ (s_1; s_2)\cdot\kappa) \longmapsto (\sigma, s_1 \cdot s_2 \cdot \kappa) \qquad \frac{\Psi;\sigma \vdash e \Downarrow v \quad \rho' = \rho_\sigma[x := v]}{\Psi \vdash (\sigma,\ (x := e)\cdot\kappa) \longmapsto (\sigma[:=\rho'], \kappa)}$$

$$\frac{\Psi;\sigma \vdash e_1 \Downarrow v_1 \qquad \Psi;\sigma \vdash e_2 \Downarrow v_2 \qquad \phi_\sigma \vdash \mathsf{store}_{ch}\, v_1 \qquad m' = m_\sigma[v_1 \overset{ch}{:=} v_2]}{\Psi \vdash (\sigma,\ ([e_1]_{ch} := e_2)\cdot\kappa) \longmapsto (\sigma[:=m'], \kappa)}$$

$$\frac{\Psi;\sigma \vdash e \Downarrow v \qquad \mathsf{is\_true}\, v}{\Psi \vdash (\sigma,\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\cdot\kappa) \longmapsto (\sigma, s_1\cdot\kappa)}$$

$$\frac{\Psi;\sigma \vdash e \Downarrow v \qquad \mathsf{is\_false}\, v}{\Psi \vdash (\sigma,\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\cdot\kappa) \longmapsto (\sigma, s_2\cdot\kappa)} \qquad \Psi \vdash (\sigma, \mathsf{skip}\cdot\kappa) \longmapsto (\sigma,\kappa)$$

$$\Psi \vdash (\sigma,\ (\mathsf{loop}\, s)\cdot\kappa) \longmapsto (\sigma, s\cdot\mathsf{loop}\, s\cdot\kappa) \qquad \Psi \vdash (\sigma, (\mathsf{block}\, s)\cdot\kappa) \longmapsto (\sigma, s\cdot\mathsf{Kblock}\,\kappa)$$

$$\frac{j \geq 1}{\Psi \vdash (\sigma, \mathsf{exit}\, 0 \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma,\kappa)}$$

$$\frac{j \geq 1}{\Psi \vdash (\sigma, \mathsf{exit}\,(n+1)\cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma, \mathsf{exit}\, n\cdot\kappa)}$$

Fig. 2. Sequential small-step relation. We omit here call and return, which are in the full technical report [4].

rules (congruence rules), which avoids induction over search rules in both the Hoare-Logic soundness proof and the compiler correctness proof.[1]

Definition 1. *A* continuation k *has a state* σ *and a* control *stack* κ*. There are sequential control operators to handle local control flow (*Kseq*, written as* $\cdot$*), intraprocedural control flow* (Kblock)*, and function-return* (Kcall)*; this last carries not only a control aspect but an activation record of its own. The control operator* Kstop *represents the safe termination of the computation.*

$$\begin{aligned} \kappa : \mathsf{control} &::= \mathsf{Kstop} \mid s\cdot\kappa \mid \mathsf{Kblock}\,\kappa \mid \mathsf{Kcall}\, xl\, f\, sp\, \rho\, \kappa \\ k : \mathsf{continuation} &::= (\sigma,\kappa) \end{aligned}$$

The sequential small-step function takes the form $\Psi \vdash k \longmapsto k'$ (see Fig. 2), and we define as usual its reflexive transitive closure $\longmapsto^*$. As in C, there is no boolean type in Cminor. In Fig. 2, the predicate $\mathsf{is\_true}\, v$ holds if v is a pointer or a nonzero integer; $\mathsf{is\_false}$ holds only on 0. A store statement $[e_1]_{ch} := e_2$ requires the corresponding store permission $\phi_\sigma \vdash \mathsf{store}_{ch}\, v_1$.

Given a control stack $\mathsf{block}\, s\cdot\kappa$, the small-step execution of the block statement $\mathsf{block}\, s$ enters that block: s becomes the next statement to execute and the control stack becomes $s \cdot \mathsf{Kblock}\,\kappa$.

[1] We have proved in Coq the equivalence of this small-step semantics with the big-step semantics of CompCert (for programs that terminate).

Exit statements are only allowed from blocks that have been previously entered. For that reason, in the two rules for exit statements, the control stack ends with $(\mathsf{Kblock}\,\kappa)$ control. A statement $(\mathsf{exit}\,n)$ terminates the $(n+1)^{th}$ enclosing block statements. In such a block, the stack of control sequences $s_1 \cdots s_j$ following the exit statement is not executed. Let us note that this stack may be empty if the exit statement is the last statement of the most enclosing block. The small-step execution of a statement $(\mathsf{exit}\,n)$ exits from only one block (the most enclosing one). Thus, the execution of an $(\mathsf{exit}\,0)$ statement updates the control stack $(\mathsf{exit}\,0 \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa)$ into κ. The execution of an $(\mathsf{exit}\ n+1)$ statement updates the control stack $(\mathsf{exit}\,(n+1) \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa)$ into $\mathsf{exit}\,n \cdot \kappa$.

Lemma 3. *If* $\Psi; \sigma \vdash e \Downarrow v$ *then* $\Psi \vdash (\sigma, (x := e) \cdot \kappa) \longmapsto k'$ *iff* $\Psi \vdash (\sigma, (x := \mathsf{Eval}\,v) \cdot \kappa)) \longmapsto k'$ *(and similarly for other statements containing expressions).*
Proof. Trivial: expressions have no side effects. A convenient property nonetheless, and not true of Leroy's original Cminor. ■

Definition 2. *A continuation* $k = (\sigma, \kappa)$ *is* stuck *if* $\kappa \neq \mathsf{Kstop}$ *and there does not exist* k' *such that* $\Psi \vdash k \longmapsto k'$.

Definition 3. *A continuation* k *is* safe *(written as* $\Psi \vdash \mathsf{safe}(k)$*) if it cannot reach a stuck continuation in the sequential small-step relation* $\longmapsto^*$.

4 Separation Logic

Hoare Logic uses triples $\{P\}\, s\, \{Q\}$ where P is a precondition, s is a statement of the programming language, and Q is a postcondition. The assertions P and Q are predicates on the program state. The reasoning on memory is inherently global. Separation Logic is an extension of Hoare Logic for programs that manipulate pointers. In Separation Logic, reasoning is local [15]; assertions such as P and Q describe properties of part of the memory, and $\{P\}\, s\, \{Q\}$ describes changes to part of the memory. We prove the soundness of the Separation Logic via a shallow embedding, that is, we give each assertion a semantic meaning in Coq. We have $P, Q : \mathsf{assert}$ where $\mathsf{assert} = \mathsf{prog} \to \mathsf{state} \to \mathsf{Prop}$. So $P\Psi\sigma$ is a proposition of logic and we say that σ satisfies P.

Assertion Operators. In Fig. 3, we define the usual operators of Separation Logic: the empty assertion **emp**, separating conjunction $*$, disjunction $\vee$, conjunction $\wedge$, implication $\Rightarrow$, negation $\neg$, and quantifier $\exists$. A state σ satisfies $P * Q$ if its footprint ϕ_σ can be split into ϕ_1 and ϕ_2 such that $\sigma[:= \phi_1]$ satisfies P and $\sigma[:= \phi_2]$ satisfies Q. We also define some novel operators such as expression evaluation $e \Downarrow v$ and base-logic propositions $\lceil A \rceil$.

O'Hearn and Reynolds specify Separation Logic for a little language in which expressions evaluate independently of the heap [15]. That is, their expressions access only the program variables and do not even have *read* side effects on the memory. Memory reads are done by a command of the language, not within expressions. In Cminor we relax this restriction; expressions can read the heap.

$$
\begin{aligned}
\mathbf{emp} &=_{\text{def}} \lambda\Psi\sigma.\ \phi_\sigma = \emptyset \\
P * Q &=_{\text{def}} \lambda\Psi\sigma.\ \exists\phi_1.\exists\phi_2.\ \ \phi_\sigma = \phi_1 \oplus \phi_2\ \wedge\ P(\sigma[:=\phi_1])\ \wedge\ Q(\sigma[:=\phi_2]) \\
P \vee Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \vee Q\sigma \\
P \wedge Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \wedge Q\sigma \\
P \Rightarrow Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \Rightarrow Q\sigma \\
\neg P &=_{\text{def}} \lambda\Psi\sigma.\ \neg(P\sigma) \\
\exists z.P &=_{\text{def}} \lambda\Psi\sigma.\ \exists z.\, P\sigma \\
\lceil A \rceil &=_{\text{def}} \lambda\Psi\sigma.\ A \qquad \text{where } \sigma \text{ does not appear free in } A \\
\mathbf{true} &=_{\text{def}} \lambda\Psi\sigma.\mathbf{True} \qquad\qquad \mathbf{false} =_{\text{def}} \lceil\mathbf{False}\rceil \\
e \Downarrow v\ &=_{\text{def}} \mathbf{emp} \wedge \lceil \mathsf{pure}(e) \rceil \wedge \lambda\Psi\sigma.\, (\Psi; \sigma \vdash e \Downarrow v) \\
\lceil e \rceil_{\mathsf{expr}} &=_{\text{def}} \exists v.\, e \Downarrow v \wedge \lceil \mathsf{is\_true}\, v \rceil \\
\mathsf{defined}(e) &=_{\text{def}} \lceil e \overset{\text{int}}{==} e \rceil_{\mathsf{expr}} \vee \lceil e \overset{\text{float}}{==} e \rceil_{\mathsf{expr}} \\
e_1 \overset{ch}{\mapsto} e_2 &=_{\text{def}} \exists v_1.\exists v_2.(e_1 \Downarrow v_1) \wedge (e_2 \Downarrow v_2) \wedge (\lambda\sigma, m_\sigma \vdash v_1 \overset{ch}{\mapsto} v_2 \wedge \phi_\sigma \vdash \mathsf{store}_{ch}\, v_1) \wedge \mathsf{defined}(v_2)
\end{aligned}
$$

Fig. 3. Main operators of Separation Logic

But we say that an expression is *pure* if it contains no Eload operators—so that it cannot read the heap.

In Hoare Logic one can use expressions of the programming language as assertions—there is an implicit coercion. We write the assertion $e \Downarrow v$ to mean that expression e is pure and evaluates to value v in the operational semantics. This is an expression of Separation Logic, in contrast to $\Psi; \sigma \vdash e \Downarrow v$ which is a judgment in the underlying logic. In a previous experiment, our Separation Logic permitted impure expressions in $e \Downarrow v$. But, this complicated the proofs unnecessarily. Having $\mathbf{emp} \wedge \lceil \mathsf{pure}(e) \rceil$ in the definition of $e \Downarrow v$ leads to an easier-to-use Separation Logic.

Hoare Logic traditionally allows expressions e of the programming language to be used as expressions of the program logic. We will define explicitly $\lceil e \rceil_{\mathsf{expr}}$ to mean that e evaluates to a true value (*i.e.* a nonzero integer or non-null pointer). Following Hoare's example, we will usually omit the $\lceil\ \rceil_{\mathsf{expr}}$ braces in our Separation Logic notation.

Cminor's integer equality operator, which we will write as $e_1 \overset{\text{int}}{==} e_2$, applies to integers or pointers, but in several cases it is "stuck" (expression evaluation gives no result): when comparing a nonzero integer to a pointer; when comparing Vundef or $\mathsf{Vfloat}(x)$ to anything. Thus we can write the assertion $\lceil e \overset{\text{int}}{==} e \rceil_{\mathsf{expr}}$ (or just write $e \overset{\text{int}}{==} e$) to test that e is a defined integer or pointer in the current state, and there is a similar operator $e_1 \overset{\text{float}}{==} e_2$.

Finally, we have the usual Separation Logic singleton "maps-to", but annotated with a chunk-type ch. That is, $e_1 \overset{ch}{\mapsto} e_2$ means that e_1 evaluates to v_1, e_2 evaluates to v_2, and at address v_1 in memory there is a defined value v_2 of the given chunk-type. Let us note that in this definition, $\mathsf{defined}(v_1)$ is implied by the third conjunct. $\mathsf{defined}(v_2)$ is a design decision. We could leave it out and have a slightly different Separation Logic.

$$\frac{P \Rightarrow P' \qquad \Gamma; R; B \vdash \{P'\}s\{Q'\} \qquad Q' \Rightarrow Q}{\Gamma; R; B \vdash \{P\}s\{Q\}} \qquad \Gamma; R; B \vdash \{P\}\mathsf{skip}\{P\}$$

$$\frac{\Gamma; R; B \vdash \{P\}s_1\{P'\} \qquad \Gamma; R; B \vdash \{P'\}s_2\{Q\}}{\Gamma; R; B \vdash \{P\}s_1; s_2\{Q\}}$$

$$\frac{\rho' = \rho_\sigma[x := v] \qquad P = (\exists v.\, e \Downarrow v \ \wedge\ \lambda\sigma.\ Q\,\sigma[:= \rho'])}{\Gamma; R; B \vdash \{P\}x := e\{Q\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \mathsf{pure}\,(e_2) \qquad P = (e \overset{ch}{\mapsto} e_2 \ \wedge\ \mathsf{defined}(e_1))}{\Gamma; R; B \vdash \{P\}[e]_{ch}{:=}e_1\{e \overset{ch}{\mapsto} e_1\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \Gamma; R; B \vdash \{P \wedge e\}s_1\{Q\} \qquad \Gamma; R; B \vdash \{P \wedge \neg e\}s_2\{Q\}}{\Gamma; R; B \vdash \{P\}\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\{Q\}}$$

$$\frac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}\mathsf{loop}\ s\{\mathbf{false}\}} \qquad \frac{\Gamma; R; Q \cdot B \vdash \{P\}s\{\mathbf{false}\}}{\Gamma; R; B \vdash \{P\}\mathsf{block}\ s\{Q\}}$$

$$\Gamma; R; B \vdash \{B(n)\}\mathsf{exit}\ n\{\mathbf{false}\}$$

$$\frac{\Gamma; R; B \vdash \{P\}s\{Q\} \qquad \text{modified vars}(s) \ \cap\ \text{free vars}(A) \ =\ \emptyset}{\Gamma; (\lambda vl.A * R(vl)); (\lambda n.A * B(n)) \vdash \{A * P\}s\{A * Q\}}$$

Fig. 4. Axiomatic Semantics of Separation Logic (without call and return)

The Hoare Sextuple. Cminor has commands to call functions, to exit (from a block), and to return (from a function). Thus, we extend the Hoare triple $\{P\}\,s\,\{Q\}$ with three extra contexts to become $\Gamma; R; B \vdash \{P\}s\{Q\}$ where:

Γ : assert describes context-insensitive properties of the global environment;
R : list val $\rightarrow$ assert is the *return environment*, giving the current function's post-condition as a predicate on the list of returned values; and
B : nat $\rightarrow$ assert is the *block environment* giving the exit conditions of each block statement in which the statement s is nested.

Most of the rules of sequential Separation Logic are given in Fig. 4. In this paper, we omit the rules for return and call, which are detailed in the full technical report. Let us note that the Γ context is used to update global function names, none of which is illustrated in this paper.

The rule for $[e]_{ch}{:=}e_1$ requires the same store permission than the small-step rule, but in Fig. 4, the permission is hidden in the definition of $e \overset{ch}{\mapsto} e_2$. The rules for $[e]_{ch}{:=}e_1$ and $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ require that e be a pure expression. To reason about an such statements where e is impure, one reasons by program transformation using the following rules. It is not necessary to rewrite the actual source program, it is only the local reasoning that is by program transformation.

$$\frac{x, y \text{ not free in } e, e_1, Q \qquad \Gamma; R; B \vdash \{P\}\, x := e;\ y := e_1;\ [x]_{ch} := y\, \{Q\}}{\Gamma; R; B \vdash \{P\} [e]_{ch} := e_1 \{Q\}}$$

$$\frac{x \text{ not free in } s_1, s_2, Q \qquad \Gamma; R; B \vdash \{P\}\, x := e; \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\, \{Q\}}{\Gamma; R; B \vdash \{P\}\, \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\, \{Q\}}$$

The statement $\mathsf{exit}\, i$ exits from the $(i+1)^{th}$ enclosing block. A block environment B is a sequence of assertions $B_0, B_1, \ldots, B_{k-1}$ such that $(\mathsf{exit}\, i)$ is safe as long as the precondition B_i is satisfied. We write $\mathsf{nil_B}$ for the empty block environment and $B' = Q \cdot B$ for the environment such that $B'_0 = Q$ and $B'_{i+1} = B_i$.

Given a block environment B, a precondition P and a postcondition Q, the axiomatic semantics of a $(\mathsf{block}\, s)$ statement consists in executing some statements of s given the same precondition P and the block environment $Q \cdot B$ (*i.e.* each existing block nesting is incremented). The last statement of s to be executed is an exit statement that yields the **false** postcondition. An $(\mathsf{exit}\, n)$ statement is only allowed from a corresponding enclosing block, *i.e.* the precondition $B(n)$ must exist in the block environment B and it is the precondition of the $(\mathsf{exit}\, n)$ statement.

Frame Rules. The most important feature of Separation Logic is the frame rule, usually written

$$\frac{\{P\}\, s\, \{Q\}}{\{A * P\}\, s\, \{A * Q\}}$$

The appropriate generalization of this rule to our language with control flow is the last rule of Fig. 4. We can derive from it a *special frame rule* for simple statements s that do not exit or return:

$$\frac{\forall R, B. (\Gamma; R; B \vdash \{P\}\, s\, \{Q\}) \qquad \text{modified vars}(s) \ \cap\ \text{free vars}(A) \ =\ \emptyset}{\Gamma; R; B \vdash \{A * P\}\, s\, \{A * Q\}}$$

Free Variables. We use a semantic notion of free variables: x is not free in assertion A if, in any two states where only the binding of x differs, A gives the same result. However, we found it necessary to use a syntactic (inductive) definition of the variables modified by a command. One would think that command c "modifies" x if there is some state such that by the time c terminates or exits, x has a different value. However, this definition means that the modified variables of `if false then` B `else` C are *not* a superset of the modified variables of C; this lack of an inversion principle led to difficulty in proofs.

Auxiliary Variables. It is typical in Hoare Logic to use auxiliary variables to relate the pre- and postconditions, e.g., the variable a in $\{x = a\}\, x := x+1\, \{x = a+1\}$. In our shallow embedding of Hoare Logic in Coq, the variable a is a Coq variable, not a Cminor variable; formally, the user would prove in Coq the proposition, $\forall a, (\Gamma; R; B \vdash \{P\} s \{Q\})$ where a may appear free in any of Γ, R, B, P, s, Q. The existential assertion $\exists z. Q$ is useful in conjunction with this technique.

Assertions about functions require special handling of these quantified auxiliary variables. The assertion that some value f is a function with precondition P and postcondition Q is written $f : \underline{\forall} x_1 \underline{\forall} x_2 \ldots \underline{\forall} x_n,\ \{P\}\{Q\}$ where P and Q

are functions from value-list to assertion, each $\underline{\forall}$ is an operator of our separation logic that binds a Coq variable x_i using higher-order abstract syntax.

Application. In the full technical report [4], we show how the Separation Logic (*i.e.* the rules of Fig. 4) can be used to prove partial correctness properties of programs, with the classical in-place list-reversal example. Such proofs rely on a set of tactics, that we have written in the tactic definition language of Coq, to serve as a proof assistant for Cminor Separation Logic proofs [3].

5 Soundness of Separation Logic

Soundness means not only that there is a model for the logic, but that the model is *the* operational semantics for which the compiler guarantees correctness! In principle we could prove soundness by syntactic induction over the Hoare Logic rules, but instead we will give a semantic definition of the Hoare sextuple $\Gamma; R; B \vdash \{P\}\, s\, \{Q\}$, and then prove each of the Hoare rules as a derived lemma from this definition.

A simple example of semantic specification is that the Hoare Logic $P \Rightarrow Q$ is defined, using the underlying logical implication, as $\forall \Psi\sigma.\ P\,\Psi\,\sigma \ \Rightarrow\ Q\,\Psi\,\sigma$. From this, one could prove soundness of the Hoare Logic rule on the left (where the $\Rightarrow$ is a symbol of Hoare Logic) by expanding the definitions into the lemma on the right (where the $\Rightarrow$ is in the underlying logic), which is clearly provable in higher-order logic:

$$\frac{P \Rightarrow Q \qquad Q \Rightarrow R}{P \Rightarrow R} \qquad\qquad \frac{\forall \Psi\sigma.(P\Psi\sigma \Rightarrow Q\Psi\sigma) \quad \forall \Psi\sigma.(Q\Psi\sigma \Rightarrow R\Psi\sigma)}{\forall \Psi\sigma.(P\Psi\sigma \Rightarrow R\Psi\sigma)}$$

Definition 4. *(a) Two states σ and σ' are* equivalent *(written as $\sigma \cong \sigma'$) if they have the same stack pointer, extensionally equivalent environments, identical footprints, and if the footprint-visible portions of their memories are the same. (b) An* assertion *is a predicate on states that is extensional over equivalent environments (in Coq it is a dependent product of a predicate and a proof of extensionality).*

Definition 5. *For any control κ, we define the assertion* **safe** κ *to mean that the combination of κ with the current state is safe:*

$$\textbf{safe}\ \kappa\ =_{\text{def}}\ \lambda\Psi\sigma.\forall\sigma'.\ (\sigma \cong \sigma'\ \Rightarrow\ \Psi \vdash \mathsf{safe}\,(\sigma', \kappa))$$

Definition 6. *Let A be a* frame*, that is, a closed assertion (i.e. one with no free Cminor variables). An assertion P* guards *a control κ in the frame A (written as $P \,\square_A\, \kappa$) means that whenever $A * P$ holds, it is safe to execute κ. That is,*

$$P \,\square_A\, \kappa\ =_{\text{def}}\ A * P\ \Rightarrow\ \textbf{safe}\ \kappa.$$

We extend this notion to say that a return-assertion R (a function from value-list to assertion) guards a return, and a block-exit assertion B (a function from block-nesting level to assertions) guards an exit:

$$R \,\boxbslash_A\, \kappa\ =_{\text{def}}\ \forall vl.R(vl) \,\square_A\, \mathsf{return}\ vl \cdot \kappa \qquad\qquad B \,\boxslash_A\, \kappa\ =_{\text{def}}\ \forall n.B(n) \,\square_A\, \mathsf{exit}\ n \cdot \kappa$$

Lemma 4. *If* $P \,\Box_A\, s_1 \cdot s_2 \cdot \kappa$ *then* $P \,\Box_A\, (s_1; s_2) \cdot \kappa$.

Lemma 5. *If* $R \,\boxed{\mathbb{R}}_A\, \kappa$ *then* $\forall s, R \,\boxed{\mathbb{R}}_A\, s \cdot \kappa$. *If* $B \,\boxed{\mathbb{B}}_A\, \kappa$ *then* $\forall s, B \,\boxed{\mathbb{B}}_A\, s \cdot \kappa$.

Definition 7 (Frame). *A* frame *is constructed from the global environment* Γ*, an arbitrary frame assertion* A*, and a statement* s*, by the conjunction of* Γ *with the assertion* A *closed over any variable modified by* s*:*

$$\mathsf{frame}(\Gamma, A, s) \;=_{\text{def}}\; \Gamma * \mathsf{closemod}(s, A)$$

Definition 8 (Hoare sextuples). *The Hoare sextuples are defined in "continuation style," in terms of implications between continuations, as follows:*

$$\Gamma; R; B \vdash \{P\}\, s\, \{Q\} \;=_{\text{def}}\; \forall A, \kappa.\\ R \,\boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,s)}\, \kappa \wedge B \,\boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,s)}\, \kappa \wedge Q \,\Box_{\mathsf{frame}(\Gamma,A,s)}\, \kappa \;\Rightarrow\; P \,\Box_{\mathsf{frame}(\Gamma,A,s)}\, s \cdot \kappa$$

From this definition we prove the rules of Fig. 4 as derived lemmas.

It should be clear from the definition—after one gets over the backward nature of the continuation transform—that the Hoare judgment specifies partial correctness, not total correctness. For example, if the statement s infinitely loops, then the continuation $(\sigma, s \cdot \kappa)$ is automatically safe, and therefore $P \,\Box_A\, s \cdot \kappa$ always holds. Therefore the Hoare tuple $\Gamma; R; B \vdash \{P\} s \{Q\}$ will hold for that s, regardless of Γ, R, B, P, Q.

Sequence. The soundness of the sequence statement is the proof that if the hypotheses $\mathsf{H}_1 : \Gamma; R; B \vdash \{P\}\, s_1\, \{P'\}$ and $\mathsf{H}_2 : \Gamma; R; B \vdash \{P'\}\, s_2\, \{Q\}$ hold, then we have to prove $\mathsf{Goal} : \Gamma; R; B \vdash \{P\}\, s_1; s_2\, \{Q\}$ (see Fig. 4). If we unfold the definition of the Hoare sextuples, H_1, H_2 and Goal become:

$$(\forall A, \kappa)\frac{R \,\boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, \kappa \qquad B \,\boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, \kappa \qquad Q \,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, \kappa}{P \,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, (s_1; s_2) \cdot \kappa}\ \mathsf{Goal}$$

We prove $P \,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, (s_1; s_2) \cdot k$ using Lemma 4:[2]

$$\dfrac{\dfrac{\dfrac{R \,\boxed{\mathbb{R}}\, k}{R \,\boxed{\mathbb{R}}\, s_2 \cdot k}\,\mathsf{Lm.\ 5} \qquad \dfrac{B \,\boxed{\mathbb{B}}\, k}{B \,\boxed{\mathbb{B}}\, s_2 \cdot k}\,\mathsf{Lm.\ 5} \qquad \dfrac{R \,\boxed{\mathbb{R}}\, k \quad B \,\boxed{\mathbb{B}}\, k \quad Q \,\Box\, k}{P' \,\Box\, s_2 \cdot k}\,\mathsf{H}_2}{P \,\Box\, s_1 \cdot s_2 \cdot k}\,\mathsf{H}_1}{P \,\Box\, (s_1; s_2) \cdot k}\ \mathsf{Lm.\ 4}$$

Loop Rule. The loop rule turns out to be one of the most difficult ones to prove. A loop continues executing until the loop-body performs an exit or return. If loop s executes n steps, then there will be 0 or more complete iterations of $n_1, n_2, \ldots$ steps, followed by j steps into the last iteration. Then either there is an exit (or return) from the loop, or the loop will keep going. But if the exit is from an inner-nested block, then it does not terminate the loop (or even this iteration). Thus we need a formal notion of when a statement exits.

[2] We will elide the frames from proof sketches by writing $\Box$ without a subscript; this particular proof relies on a lemma that $\mathsf{closemod}(s_1, \mathsf{closemod}((s_1; s_2), A)) = \mathsf{closemod}((s_1; s_2), A)$.

Consider the statement $s = \mathsf{if}\, b\, \mathsf{then}\, \mathsf{exit}\, 2\, \mathsf{else}\, (\mathsf{skip}; x := y)$, executing in state σ. Let us execute n steps into s, that is, $\Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^n (\sigma', \kappa')$. If n is small, then the behavior should not depend on κ; only when we "emerge" from s is κ important. In this example, if $\rho_\sigma b$ is a true value, then as long as $n \leq 1$ the statement s can *absorb* n steps independent of κ; if $\rho_\sigma b$ is a false value, then s can absorb up to 3 steps. To reason about absorption, we define the concatenation $\kappa_1 \circ \kappa_2$ of a control prefix κ_1 and a control κ_2 as follows:

$$\begin{array}{ll} \mathsf{Kstop} \circ \kappa =_{\text{def}} \kappa & (\mathsf{Kblock}\, \kappa') \circ \kappa =_{\text{def}} \mathsf{Kblock}\, (\kappa' \circ \kappa) \\ (s \cdot \kappa') \circ \kappa =_{\text{def}} s \cdot (\kappa' \circ \kappa) & (\mathsf{Kcall}\, xl\, f\, sp\, \rho\, \kappa') \circ \kappa =_{\text{def}} \mathsf{Kcall}\, xl\, f\, sp\, \rho\, (\kappa' \circ \kappa) \end{array}$$

Kstop is the empty prefix; $\mathsf{Kstop} \circ \kappa$ does not mean "stop," it means κ.

Definition 9 (absorption). *A statement s in state σ* $\mathsf{absorbs}$ *n steps (written as* $\mathsf{absorb}(n, s, \sigma)$*) iff* $\forall j \leq n. \exists \kappa_{\text{prefix}}. \exists \sigma'.\ \forall \kappa.\ \Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^j (\sigma', \kappa_{\text{prefix}} \circ \kappa)$.

Example 1. An exit statement by itself absorbs no steps (it immediately uses its control-tail), but $\mathsf{block}\, (\mathsf{exit}\, 0)$ can absorb the 2 following steps:
$\Psi \ \vdash\ (\sigma, \mathsf{block}\, (\mathsf{exit}\, 0) \cdot \kappa) \longmapsto (\sigma, \mathsf{exit}\, 0 \cdot \mathsf{Kblock}\, \kappa) \longmapsto (\sigma, \kappa)$

Lemma 6. *1.* $\mathsf{absorb}(0, s, \sigma)$.
2. $\mathsf{absorb}(n+1, s, \sigma) \ \Rightarrow\ \mathsf{absorb}(n, s, \sigma)$.
3. If $\neg\mathsf{absorb}(n, s, \sigma)$*, then* $\exists i < n. \mathsf{absorb}(i, s, \sigma) \wedge \neg\mathsf{absorb}(i+1, s, \sigma)$*. We say that s absorbs at most i steps in state σ.*

Definition 10. *We write $(s;)^n s'$ to mean* $\underbrace{s; s; \ldots; s;}_{n}\ s'$.

Lemma 7. $$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} (s;)^n \mathsf{loop}\, \mathsf{skip} \{\mathbf{false}\}}$$

Proof. For $n = 0$, the infinite-loop $(\mathsf{loop}\, \mathsf{skip})$ *satisfies any precondition for partial correctness. For $n+1$, assume κ, $R \,\boxed{\mathsf{R}}\, \kappa$, $B \,\boxed{\mathsf{B}}\, \kappa$; by the induction hypothesis (with $R \,\boxed{\mathsf{R}}\, \kappa$ and $B \,\boxed{\mathsf{B}}\, \kappa$) we know $I \,\square\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$. We have $R \,\boxed{\mathsf{R}}\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ and $B \,\boxed{\mathsf{B}}\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ by Lemma 5. We use the hypothesis $\Gamma; R; B \vdash \{I\} s \{I\}$ to augment the result to $I \,\square\, (s; (s;)^n \mathsf{loop}\, \mathsf{skip}) \cdot \kappa$.* ■

Theorem 1. $$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} \mathsf{loop}\, s \{\mathbf{false}\}}$$

Proof. Assume κ, $R \,\boxed{\mathsf{R}}\, \kappa$, $B \,\boxed{\mathsf{B}}\, \kappa$. To prove $I \,\square\, \mathsf{loop}\, s \cdot \kappa$, assume σ and $I\sigma$ and prove $\mathsf{safe}\, (\sigma, \mathsf{loop}\, s \cdot \kappa)$*. We must prove that for any n, after n steps we are not stuck. We unfold the loop n times, that is, we use Lemma 7 to show* $\mathsf{safe}\, (\sigma, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa)$*. We can show that if this is safe for n steps, so is $\mathsf{loop}\, s \cdot \kappa$ by the principle of absorption. Either s absorbs n steps, in which case we are done; or s absorbs at most $j < n$ steps, leading to a state σ' and a control (respectively) $\kappa_{prefix} \circ (s;)^{n-1} \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ or $\kappa_{prefix} \circ \mathsf{loop}\, s \cdot \kappa$. Now, because s cannot absorb $j+1$ steps, we know that either κ_{prefix} is empty (because s has terminated normally) or κ_{prefix} starts with a* return *or* exit*, in which case we*

escape (resp. past the loop skip *or the* loop s*) into* κ*. If* κ_{prefix} *is empty then we apply strong induction on the case* $n - j$ *steps; if we escape, then* (σ', κ) *is safe iff* $(\sigma, \mathsf{loop}\, s \cdot \kappa)$ *is safe. (For example, if* $j = 0$*, then it must be that* $s =$ return *or* $s =$ exit *, so in one step we reach* $\kappa_{prefix} \circ (\mathsf{loop}\, s \cdot \kappa)$ *with* $\kappa_{prefix} =$ return *or* $\kappa_{prefix} =$ exit *.)* ■

6 Sequential Reasoning About Sequential Features

Concurrent Cminor, like most concurrent programming languages used in practice, is a sequential programming language with a few concurrent features (locks and threads) added on. We would like to be able to reason about the sequential features using purely sequential reasoning. If we have to reason about all the many sequential features without being able to assume such things as determinacy and sequential control, then the proofs become much more difficult.

One would expect this approach to run into trouble because critical assumptions underlying the sequential operational semantics would not hold in the concurrent setting. For example, on a shared-memory multiprocessor we cannot assume that (x:=x+1; x:=x+1) has the same effect as (x:=x+2); and on any real multiprocessor we cannot even assume *sequential consistency*—that the semantics of n threads is some interleaving of the steps of the individual threads.

We will solve this problem in several stages. Stage 1 of this plan is the current paper. Stages 2, 3, and 4 are work in progress; the remainder is future work.

1. We have made the language, the Separation Logic, and our proof extensible: the set of control-flow statements is fixed (inductive) but the set of straight-line statements is extensible by means of a parameterized module in Coq. We have added to each state σ an *oracle* which predicts the meaning of the extended instruction (but which does nothing on the core language). All the proofs we have described in this paper are on this extensible language.
2. We define spawn, lock, and unlock as extended straight-line statements. We define a concurrent small-step semantics that assumes noninterference (and gets "stuck" on interference).
3. From this semantics, we calculate a single-thread small-step semantics equipped with the oracle that predicts the effects of synchronizations.
4. We define a Concurrent Separation Logic for Cminor as an extension of the Sequential Separation Logic. Its soundness proof uses the sequential soundness proof as a lemma.
5. We will use Concurrent Separation Logic to guarantee noninterference of source programs. Then (x:=x+1; x:=x+1) *will* have the same effect as (x:=x+2).
6. We will prove that the Cminor compiler (CompCert) compiles each footprint-safe source thread into an equivalent footprint-safe machine-language thread. Thus, noninterfering source programs will produce noninterfering machine-language programs.

7. We will demonstrate, with respect to a formal model of weak-memory-consistency microprocessor, that noninterfering machine-language programs give the same results as they would on a sequentially consistent machine.

7 The Machine-Checked Proof

We have proved in Coq the soundness of Separation Logic for Cminor. Each rule is proved as a lemma; in addition there is a main theorem that if you prove all your function bodies satisfy their pre/postconditions, then the program "call main()" is safe. We have informally tested the adequacy of our result by doing tactical proofs of small programs [3].

Lines	Component
41	Axioms: dependent unique choice, relational choice, extensionality
8792	Memory model, floats, 32-bit integers, values, operators, maps (exactly as in CompCert [12])
4408	Sharable permissions, Cminor language, operational semantics
462	Separation Logic operators and rules
9874	Soundness proof of Separation Logic

These line counts include some repetition of specifications (between Modules and Module Types) in Coq's module system.

8 Conclusion

In this paper, we have defined a formal semantics for the language Cminor. It consists of a big-step semantics for expressions and a small-step semantics for statements. The small-step semantics is based on continuations mainly to allow a uniform representation of statement execution. The small-step semantics deals with nonlocal control constructs (return, exit) and is designed to extend to the concurrent setting.

Then, we have defined a Separation Logic for Cminor. It consists of an assertion language and an axiomatic semantics. We have extended classical Hoare triples to sextuples in order to take into account nonlocal control constructs. From this definition of sextuples, we have proved the rules of axiomatic semantics, thus proving the soundness of our Separation Logic.

We have also proved the semantic equivalence between our small-step semantics and the big-step semantics of the CompCert certified compiler, so the Cminor programs that we prove in Separation Logic can be compiled by the CompCert certified compiler. We plan to connect a Cminor certified compiler directly to the small-step semantics, instead of going through the big-step semantics.

Small-step reasoning is useful for sequential programming languages that will be extended with concurrent features; but small-step reasoning about nonlocal control constructs mixed with structured programming (loop) is not trivial. We have relied on the determinacy of the small-step relation so that we can define concepts such as $\mathsf{absorb}(n, s, \sigma)$.

References

1. The Coq proof assistant, `http://coq.inria.fr`
2. American National Standard for Information Systems – Programming Language – C: American National Standards Institute (1990)
3. Appel, A.W.: Tactics for separation logic (January 2006), `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`
4. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor (extended version). Technical Report RR 6138, INRIA (March 2007), `https://hal.inria.fr/inria-00134699`
5. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)
6. Blazy, S., Leroy, X.: Formal verification of a memory model for C-like imperative languages. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 280–299. Springer, Heidelberg (2005)
7. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL'05, pp. 259–270. ACM Press, New York (2005)
8. Dargaye, Z.: Décurryfication certifiée. JFLA (Journées Françaises des Langages Applicatifs), 119–133 (2007)
9. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: POPL'01, January 2001, pp. 14–26. ACM Press, New York (2001)
10. Klein, G., Tuch, H., Norrish, M.: Types, bytes, and separation logic. In: POPL'07, January 2007, pp. 97–108. ACM Press, New York (2007)
11. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM'05. IEEE Conference on Software Engineering and Formal Methods, IEEE Computer Society Press, Los Alamitos (2005)
12. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL'06, pp. 42–54. ACM Press, New York (2006)
13. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning 5(4), 461–492 (1989)
14. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: POPL'06, January 2006, pp. 320–333. ACM Press, New York (2006)
15. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
16. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
17. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)