УДК 004.052.42

Towards automated error localization in C programs with loops*

Kondratyev D.A. (A.P. Ershov Institute of Informatics Systems SB RAS)

Promsky A.V. (A.P. Ershov Institute of Informatics Systems SB RAS)

The most recent trends in the C-light verification system are MetaVCG, semantic labels appropriate for verification condition (VC) explanation and symbolic method of definite iterations. MetaVCG takes a C-light program together with some Hoare's logic and produces on-the-fly a VC generator (VCG), which in turn processes the input program. Hoare's logic for definite iterations is a good choice if we try to get rid of loop invariants. Finally, if a theorem prover was unable to validate some VCs we could follow two ways. Obviously, we could revise/enrich specifications or/and underlying proof theory to prove the truth of VCs. Or, perhaps, we could concentrate upon establishment of falsity, which meant there were errors in annotated program. This is where semantic labels play crucial role providing some natural language comments about wrong VC as well as a back-trace to the error location. The newly developed ACL2 heuristics to prove VC falsity is the main theme of this paper.

Keywords: *deductive verification, semantic label, error localization, C-light, automated theorem proof, C-lightVer, ACL2, MetaVCG, symbolic method of verification of definite iterations, proof* strategy

1. Introduction

The C-light project [12] corresponds to the mainstream architecture of modern verification systems. It uses translation into an intermediate language (here, C-kernel) allowing to smooth over some hard corners of deductive verification. In order to improve efficiency we prefer domain specific verification condition (VC) generation, which means different generators for different program classes. Traditional approach implies manual reprogramming of VC generator (VCG). Instead, we adapted the MetaVCG approach of Moriconi and Schwarts [13]. For a given axiomatic system the MetaVCG automatically constructs an ordinary generator. The C-lightVer system is the implementation of the C-light project.

Are there any verification problems that cannot be solved by a two-level scheme or by

*This research is partially supported by RFBR grant 17-01-00789.

MetaVCG? Indeed, there are. The loop invariants, for example. Another approach adapted in our project, so called symbolic verification of definite iterations [14], proposes solution for certain class of cycles over data structures. The loop execution is modeled symbolically by replacement function `rep`. Some results of such adaptation were discussed in [6, 8].

At the moment, the main theorem prover for C-light is ACL2 [5]. The classic induction in ACL2 is not powerful enough to handle VCs with replacement function `rep`, so the proof strategies are proposed in [8, 11].

The traditional deductive verification works ideally for a priori correct Hoare triple with true VCs. If some VCs are false the user must analyze the prover signals to understand what went wrong and to localize possible errors. An easy task for toy examples which becomes a real problem for real programs. Denney and Fisher developed the semantic labeling approach [3]. For every VC it provides the proof protocol (in axiomatic semantics) as well as localizing information up to the level of separate terms. A natural language explanation can be generated from such protocol. The MetaVCG approach allowed us to easily introduce semantic labels in our axiomatic systems [6, 7, 12].

Now, after this background overview let us address the current problematic task, error localization for programs with loops. And again, situation is clear when ACL2 is able to discover truth or falsity. But often the answer is `"unknown"`. Instead of trying to satisfy a VC we can use some strategies to check unsatisfiability. Since all variables in ACL2 are implicitly universally quantified, the existence quantifier appears in the negated VC. Thus, our previous strategies [8, 11] fail here and we need revised ones. Another requirement — the `unsat` strategies must work for loops with abrupt termination (i.e. in presence of `break` statement). Possible solution of this problem is discussed in this paper.

Related work. Some proof strategies deserve mention. For example, the system `ACL2(ml)` [4] bases upon two methods. The first one is proof pattern recognition by means of statistical machine learning. The second one is symbolic searching for analogous lemma. However, the underlying theories may vary deeply, so machine learning is not very suitable for VC proof.

The SL-resolution is another well-known strategy [10]. Its inherent problem consists in necessary construction of useless resolvents. To oppose the growth of disjunct set the connection graph method was proposed [15]. In comparison to it, our strategy aims at the structure of literals, not disjuncts.

The goals of Constraint Logic Transformation project [2] remind ours. In the same time

their strategies are suitable for predicate processing.

2. Methods used in C-light project

MetaVCG. The metagenerator takes proof rules and axioms as its input. Technically all of them represent patterns to be matched against C code [9]. The pattern language incorporates the first order logics and the C grammar. Expressions can contain nonterminal symbols, like uninterpreted predicate symbols or "fragment variables" [13] denoting code snippets. Affiliation of metadata with certain class in the pattern language is explicit [9]. For example, construction *any_code*(S) can be matched against any sequence (including empty) of C statements, construction *any_predicate*(P) can be matched against any predicate of specification language. Let vector $v = \langle v_1 \dots v_k \rangle$. Let each $expr_j (1 \le j \le k)$ is result of replacement of all occurrences of term *vector_element* in the expression *expr* by v_j. Then construct *vector_substitution*$(T,\ vec,\ expr)$ denotes the simultaneous replacement of all occurrences of each $v_j (1 \le j \le k)$ in the formula T by expression $expr_j$.

Semantic labeling. The idea of Denney and Fischer [3] consists in adding of semantic labels to the proof rules. Labels explain the result of rule application. We also use notation $\ulcorner t \urcorner^l$, which means that the term t is decorated with label l. Labels themselves take form $c(o, n)$, where c is a concept (the term role), o is a line range (in the source program) and n is an auxiliary information.

In contrast to the original idea, the labels in our VCs form hierarchy more suitable to explanation generation [6]. We perform the depth search in the label tree and for each label its common text is expanded by corresponding pattern filled by line numbers. The text patterns for every label concept are similar to the C format strings, they are also fed to MetaVCG.

In order to support arbitrary label concepts a special construction `(label t c)` was added to the language of proof rules [7]. Here, `t` is a term decorated by label and `c` is a string (label type).

Symbolic method for the definite iterations. Consider a program fragment of the form `for x in S do v := body(v, x) end`, where S is a data structure, x is a variable of type "element of S", v is tuple of the loop variables excluding x, *body* represents the loop body which does not alter x and terminates for every $x \in S$. The ways of modification of S are quite restricted. The loop body can only contain assignments, the `if` statements (perhaps nested)

ant the `break` statements. We call such `for` loops definite iterations. Let v_0 be the tuple of values of variables from v at the loop entry point. Result of the whole definite iteration can be expressed by replacement operation $rep(v, S, body, n)$, where $rep(v, S, body, 0) = v_0$, $rep(i, v, S, body) = body(rep(i-1, v, S, body), s_i)$ for all $i = 1, 2, \ldots, n$. If the `break` statement was executed during the i-th iteration of the body, the whole definite iteration continues, though v does not change anymore: $\forall j (i \le j \le n)\ rep(i, v, S, body) = rep(j, v, S, body)$.

The MetaVCG allowed us to combine ideas of definite iterations and semantics labels in the form of the following pattern:

```
{P} prog {(vector_substitution(Q, v,
                        (label rep_iter rep(n, v, S, body).vector_element))}
|- {any_predicate(P)} any_code(prog)
for(int_var(i) = 0; int_var(i) < int_var(n); int_var(i)++)
admissible_construct(i, n, v, S, body)
{any_predicate(Q)}
```

where $admissible\_construct(i, n, v, S, body)$ corresponds to an admissible body of definite iteration, int_var corresponds to an integer variable. The construct $vector\_substitution(Q,\ v,\ rep(n,\ v,\ S,\ body).vector\_element)$ denotes the simultaneous replacement of all occurrences of each $v_t (1 \le t \le length(v))$ in the formula Q by $rep(n,\ v,\ S,\ body).v_t$. The recursive definition of admissible construct is described in [8]. The algorithms of matching these patterns and program constructs have been implemented in the C-lightVer system [6–9]. The inference rule for downward iteration is defined similarly.

Automated generation of replacement operation. The replacement operation generation is based on translation [8] of loop body constructs into ACL2. Consider, for example, construction $b*$:

$$(b* \ (\ldots (var\ expr) \ldots)\ result)$$

Expression $(var\ expr)$ denotes binding of variable var with the value of expression $expr$, which may depend on previously bound variables. The value of $b*$ is equal to value of $result$, which also can depend on bound variables. Values of variables from v correspond to values of members of structure fr of type $frame$. So, to model modification of some variable in v we bind object fr with new object which differs from the old one in the corresponding field. The abrupt termination is modeled by truth of boolean member *loop-break* of object fr. Instruction

break is modeled by the following binding: $((when\ t)\ fr)$. Since condition *when* is true, such binding interrupts current block $b*$ and returns fr.

3. Proof strategy for formulas with replacement operation

The arguments of this strategy are implication ψ containing expression $rep(n, ...)$ and the premise ϕ.

In the beginning we try to prove formula

$$\phi \rightarrow rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}1)$$

by induction on n. If ACL2 succeeds, we add (ψ-*lemma-1*) to the underlying theory. This lemma means that premise ϕ of implication ψ represents situation when the loop is abruptly terminated. Then we try to prove ψ using (ψ-*lemma-1*) and induction on n.

If ACL2 failed to prove (ψ-*lemma-1*), we address to the formula

$$\phi \rightarrow \neg rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}2)$$

by induction on n. Again, if ACL2 validates (ψ-*lemma-2*) we add it to our theory in order to be used in the proof of ψ.

This strategy resembles one described in [8]. Indeed, both of them use the value of *loop-break* field. They are automatic. Finally, they are heuristics.

But differences also take place. First, this strategy is applied to any implication containing `rep`, whereas strategy from [8] analyses program postconditions only. Second, the latter one generates lemmas in the form of conjunction. The first conjunct is a VC and the second one establishes equality of antecedent from postcondition to the value of *loop-break*. Thus, lemmas generated in [8] have rather more complex structure.

4. UNSAT strategy for VCs

The argument ω of this strategy contains expression $rep(n, \ldots)$ and has the following form:

$$\forall x_1 \ldots x_n\ (\phi_1(x_1 \ldots x_n) \rightarrow \psi_1(x_1 \ldots x_n)) \wedge \ldots \wedge (\phi_m(x_1 \ldots x_n) \rightarrow \psi_m(x_1 \ldots x_n))$$

In ACL2 all variables of ω are implicitly universally quantified. Note that every ϕ_i or ψ_i is not forced to depend on all variables $x_1 \ldots x_n$. But for simplicity we imply such dependence.

Now let us prove $\neg\omega$:

$$(\exists x_1 \dots x_n \ (\phi_1(x_1 \dots x_n) \wedge \neg\psi_1(x_1 \dots x_n))) \vee \dots \vee (\exists x_1 \dots x_n \ (\phi_m(x_1 \dots x_n) \wedge \neg\psi_m(x_1 \dots x_n)))$$

Let for each $i(1 \leq i \leq m)$ $T_i \equiv \exists x_1 \dots x_n \ (\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$. To prove $\neg\omega$ it is sufficient to prove an arbitrary $T_i(1 \leq i \leq m)$. If we can find such T_i the answer of strategy is `VC is false`. Otherwise, the answer is `Unknown.`

Now, to the proof of T_i. Obviously truth of formula

$$T_i' \equiv (\exists x_1 \dots x_n \ \phi_i(x_1 \dots x_n)) \wedge (\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \rightarrow \neg\psi_i(x_1 \dots x_n)))$$

is sufficient to prove T_i. Let us denote subformula $\exists x_1 \dots x_n \ \phi_i(x_1 \dots x_n)$ as U_i and the right subformula $\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \rightarrow \neg\psi_i(x_1 \dots x_n))$ as V_i. We need to prove both U_i and V_i.

The process begins with interactive proof of U_i. Admittedly, sounds like undesirable step away from automated verification but the reasons are as follows. First, we use the weakest precondition calculus [8]. Therefore, U_i cannot contain replacement operation. Only the specification functions that are known to user take place in U_i. Second, almost all automatic proof assistants have problems with existence quantifier. Third, our heuristics rests upon hypothesis of simple antecedents and complex consequents in implications that were generated by the wp-calculus. Based on this assumption, we try to automatize a more complex proof of V_i. If user cannot validate U_i we suppose that the whole T_i is false.

If user confirms U_i, the proof of V_i starts. There are two possibilities. First, V_i may be free of function `rep`. We simply pass this universally quantified formula to ACL2. Second, V_i contains expression $rep(n, \dots)$. Anyway, since user does not know definition of `rep`, only the automated attempts are possible. The `break` statement can cause some trouble here because it complicates definition of `rep`. In this case, we address the strategy from Section 3.

5. Example

The goal of this experiment is to demonstrate execution of both strategies. This case study is also described in our on-line repository [1]. We deliberately made an error in the following function:

```
1. /*@ requires (0 < n) && (n <= len(a));
2.     ensures (grt-eql-cnt(n, key, a) == 0 ==> \result == 0) &&
```

```
3.              (grt-eql-cnt(n, key, a) > 0 ==> \result == 1)
4. */
5. int grt_eql_key(int n, int key, int a[]){
6.    int i, result = 0;
7.    for (i = 0 ; i < n; i++){
8.        if (a[i] < key){result = 1; break;}}
9.    return result;}
```

We suppose the reader is familiar with ACSL format of specifications. The logical function *grt-eql-cnt* counts the number of array elements greater or equal to `key`. Its definition is given in Appendix A. The program should look for array element that is greater than or equal to `key` and should return `1` or `0` according to the specification. Thus, the error is the use of operator `<` instead of operator `>=` in if-condition of loop body. The result of intermediate translation into C-kernel looks like:

```
 5. int grt_eql_key(int n, int key, int a[]){
 6.    /* begin changes Dec3 1 7-8 */
 7.    auto int i;
 8.    auto int result = 0;
 9.    /* end changes */
10.    for (i = 0 ; i < n; i++){
11.        if (a[i] < key){result = 1; break;}}
12.    return result;}
```

Note that neither specification nor definite iteration is modified by translator. The string `"begin changes Dec3 1 7-8"` stores data for the error localization protocol [12]. Of course, it did not appear because we knew about intentional error. It is inherent mechanism of our translation stage. In this particular case it declares that translation rule `Dec3` for declarations [12] was used and strings `7-8` are its result. Due to the symbolic method VCG produces single VC (*vc*-1):

$$\forall n, key, a$$
$$((\ulcorner 0 < n \wedge n \leq len(a) \urcorner^{ass\_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$\ulcorner grt\text{-}eql\text{-}cnt(n, key, a) = 0 \urcorner^{ass\_post(2)} \rightarrow$$
$$\ulcorner \ulcorner rep(n, key, a, 0).result \urcorner^{rep\_iter(10-11)} = 0 \urcorner^{ens\_post(2)})$$
$$\wedge$$
$$(\ulcorner 0 < n \wedge n \leq len(a) \urcorner^{ass\_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$\ulcorner grt\text{-}eql\text{-}cnt(n, key, a) > 0 \urcorner^{ass\_post(3)} \rightarrow$$
$$\ulcorner \ulcorner rep(n, key, a, 0).result \urcorner^{rep\_iter(10-11)} = 1 \urcorner^{ens\_post(3)})$$

where $IntArr$ is set of integral arrays. Function rep is defined in Appendix B. Semantic label ass_pre denotes hypothesis from precondition, ass_pre denotes hypothesis from postcondition, ens_post denotes goal from postcondition, rep_iter denotes substitution of replacement function [7]. As expected, vc-1 cannot be proved by SAT strategies, like those from [8, 11]. It is time to use our UNSAT strategy.

Formula vc-1 is a conjunction of two implications. Each of them uses function rep. The formula U_1 for the first conjunct ϕ looks like:

$$\exists n, key, a$$
$$0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$grt\text{-}eql\text{-}cnt(n, key, a) = 0$$

Since the user knows definition of *grt-eql-cnt*, he can prove U_1 in interactive mode. Therefore, formula V_1 for conjunct ϕ appears:

$$\forall n, key, a$$
$$\ulcorner 0 < n \wedge n \leq len(a) \urcorner^{ass\_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$\ulcorner grt\text{-}eql\text{-}cnt(n, key, a) = 0 \urcorner^{ass\_post(2)} \rightarrow$$
$$\ulcorner \ulcorner rep(n, key, a, 0).result \urcorner^{rep\_iter(10-11)} \neq 0 \urcorner^{ens\_post(2)}$$

This part of the proof is automatic. The presence of `break` in the loop body complicates things, so strategy from Section 3 enters on the scene. The corresponding V_1-*lemma*-1 is as follows:

$$\forall n, key, a$$
$$0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$grt\text{-}eql\text{-}cnt(n, key, a) = 0 \rightarrow$$
$$rep(n, key, a, 0).loop\text{-}break$$

This lemma deserves further explanation. Since $0 < n$, the loop body is executed at least once. Amount of elements of a greater or equal to key is also zero. Thus, all elements in sub-array

$[0 : n-1]$ are less than key. Control expression of the if statement contains wrong operator, so for such array the $break$ statement is executed. V_1-$lemma$-1 was proved automatically in ACL2 by induction on n and was added to the theory. Its `lisp` definition is given in Appendix C.

Finally, execution of `break` means that $result = 1$. So, V_1-$lemma$-1 contributes in automatic proof of V_1. The `lips` definition of V_1 is given in Appendix D. As a result vc-1 is false and explanation for V_1 is produced. Let us consider this explanation.

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that postcondition goal from line 2
    with substitution loop effect from lines 7-8 by rep
does not hold.
```

The error localization protocol [12] analyses semantic labels in V_1 and results of UNSAT strategy to generate the text above. Actually, labels in VCs contain string ranges of C-kernel program. This is where the commented information about translation rules proves to be useful. The location of possible error can be retranslated back to C-light program [12].

6. Conclusion

Many papers tend to demonstrate successful experiments avoiding the situation when verification fails. To fill this gap we discussed here some ideas about error localization in the C-light project. In case when the prover can confirm neither truth nor falsity, the traditional response if straightforward. User tries to reinforce the underlying theory in attempt to successfully reprove VCs. However, we do not discard bad scenario and simultaneously we try to check whether VCs are actually false. Another point of our current interest is verification of loops with `break` statement. We obtained some results in our experiments:

1. We devised the UNSAT strategy. It generates formula whose truth automatically implies falsity of original VC. At the moment such formula involves either interactive and automated proof.
2. We also propose the SAT strategy for VCs with operation `rep`. If abrupt termination of the loop is described in formula premises, a corresponding lemma will be added to the

underlying theory.

These strategies are the new contribution relative to previous research [6–8, 11]. Based on these methods, we conducted some experiments on error localization. We can also mention successful verification [6, 8] of function `asum` from the interface BLAS. The more complex data structures and the other functions from BLAS will be considered in future work.

References

1. Automated Error Localization in C Programs. URL: https://bitbucket.org/Kondratyev/verify-c-light. Last accessed 30 Apr 2019.
2. De Angelis E., Fioravanti F., Pettorossi A., Proietti M.: Verification of Imperative Programs by Constraint Logic Program Transformation // Electronic Proceedings in Theoretical Computer Science. Vol. 129. P. 186–210.
3. Denney E., Fischer B. Explaining Verification Conditions // Lecture Notes in Computer Science. B.: Springer-Verlag, 2008. Vol. 5140. P. 145–159.
4. Heras J., Komendantskaya E., Johansson M., Maclean E. Proof-Pattern Recognition and Lemma Discovery in ACL2 // Lecture Notes in Computer Science. B.: Springer-Verlag, 2013. Vol. 8312. P. 389–406.
5. Hunt W. A., Kaufmann M., Moore J. S., Slobodova A. Industrial hardware and software verification with ACL2 // Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 2017. Vol. 375. No. 2104. Article Number 20150399.
6. Kondratyev D. Implementing the Symbolic Method of Verification in the C-Light Project // Lecture Notes in Computer Science. Cham: Springer International Publishing AG, 2018. Vol. 10742. P. 227–240.
7. Kondratyev D.A. The extension of the C-light project using symbolic verification method of definite iterations // XVII All-Russian Conf. of Young Scientists on Mathematical Modeling and Information Technology. Computational technologies. 2017. Vol. 22. P. 44-59. (In Russian)
8. Kondratyev D. A., Maryasov I. V., Nepomniaschy V. A. The Automation of C Program Verification by Symbolic Method of Loop Invariants Elimination // Modeling and Analysis of Information Systems. 2018. Vol. 25. No. 5. P. 491–505. (In Russian)
9. Kondratyev D. A., Promsky. A. V. Developing a self-applicable verification system. Theory and practice // Automatic Control and Computer Sciences. 2015. Vol. 49. No. 7. P. 445–452.
10. Kowalski R., Kuehner D. Linear Resolution with Selection Function // Artificial Intelligence. 1971. Vol. 2. No. 3–4. P. 227–260.
11. Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A. Invariant Elimination of Definite Iterations over Arrays in C Programs Verification // Modeling and Analysis of Information Systems. 2017. Vol. 24. No. 6. P. 743–754.
12. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program

Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. 2014. Vol. 48. No. 7. P. 407–414.

13. Moriconi M., Schwarts R.L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lecture Notes in Computer Science. B.: Springer-Verlag, 1981. Vol. 115. P. 363–377.
14. Nepomniaschy V.A. Symbolic method of verification of definite iterations over altered data structures // Programming and Computer Software. 2005. Vol. 31. No. 1. P. 1–9.
15. Siekmann J., Wrightson G. An Open Research Problem: Strong Completeness of R. Kowalski's Connection Graph Proof Procedure // Lecture Notes in Computer Science. B.: Springer-Verlag, 2002. Vol. 2408. P. 231–252.

A. The ACL2 definition of *grt-eql-cnt*

```
(define grt-eql-cnt ((n integerp) (key integerp) (a integer-listp))
    :guard-hints (("Goal" :induct (dec-induct n)))
    :returns (result natp :hints (("Goal" :induct (dec-induct n))))
    (b* (   (n (nfix n))
            (key (ifix key))
            (a (integer-list-fix a))
            ((when (zp n)) 0)
            ((when (< (len a) n)) 0))
        (if (<= key (nth (- n 1) a))
            (+ 1 (grt-eql-cnt (- n 1) key a))
            (grt-eql-cnt (- n 1) key a)))
    ///
    (fty::deffixequiv grt-eql-cnt))
```

Due to recursive nature of *grt-eql-cnt*, induction is preferable when it comes to the proof of statements containing this function.

B. The ACL2 definition of function *rep*

```
(fty::defprod frame ((loop-break booleanp) (i integerp) (result integerp)))

(fty::defprod envir ((lower-bound integerp) (key integerp) (a integer-listp)))

(define frame-init ((i integerp) (result integerp))
    :returns (fr frame-p)
    (make-frame :loop-break nil
        :i i
        :result result)
    ///
```

```
    (fty::deffixequiv frame-init))

(define envir-init ((lower-bound integerp) (key integerp) (a integer-listp))
    :returns (env envir-p)
    (make-envir :lower-bound lower-bound
        :key key
        :a a)
    ///
    (fty::deffixequiv envir-init))

(define rep ((iteration natp) (env envir-p) (fr frame-p))
    :measure (nfix iteration)
    :verify-guards nil
    :returns (upd-fr frame-p)
    (b* (   (iteration (nfix iteration))
            (env (envir-fix env))
            (fr (frame-fix fr))
            ((when (zp iteration)) fr)
            (fr (rep (- iteration 1) env fr))
            ((when (frame->loop-break fr)) fr)
            (fr (if (<  (nth
                            (- (+ iteration (envir->lower-bound env)) 1)
                            (envir->a env))
                        (envir->key env))
                    (b* (   (fr (change-frame fr :result 1))
                            (fr (change-frame fr :loop-break t))
                            ((when t) fr)) fr)
                    (b* ((fr fr)) fr)))
            ((when (frame->loop-break fr)) fr)
            (fr (change-frame fr
                     :i (+ iteration (envir->lower-bound env)))))
        fr))
```

C. The ACL2 definition of V_1-*lemma*-1

```
(defrule v-1-lemma-1
    (implies (and (< 0 n) (<= n (len a)) (integerp n) (integerp key) (integer-listp a)
            (= 0 (grt-eql-cnt n key a)))
        (frame->loop-break
            (rep n (envir-init 0 key a) (frame-init 0 0))))
    :enable (grt-eql-cnt envir-init frame-init rep)
    :hints (("Goal" :induct (dec-induct n))))
```

Construction of the form

```
    :hints (("Goal" :induct (dec-induct n)))
```

prompts ACL2 to use induction on n.

D. The ACL2 definition of lemma V_1

```
(defrule v-1
    (implies (and (< 0 n) (<= n (len a)) (integerp n) (integerp key) (integer-listp a)
            (= 0 (grt-eql-cnt n key a)))
        (not (= (frame->result
                    (rep n (envir-init 0 key a) (frame-init 0 0))) 0)))
    :enable (grt-eql-cnt envir-init frame-init rep v-1-lemma-1)
    :hints (("Goal" :induct (dec-induct n))))
```