Bull. Nov. Comp. Center, Comp. Science, 31 (2010), 139–154
© 2010 NCC Publisher

Guided tour inside F@BOOL@: a case-study for a SAT-based verifying compiler*

Nikolay Shilov, Eugene Bodin, Svetlana Shilova

Abstract. A verifying compiler is a system program that translates programs written by an user from a high-level language into equivalent executable programs, and besides, proves (verifies) mathematical statements specified by the human about the properties of the programs being translated. The purpose of the F@BOOL@ project is to develop a transparent for users, compact and portable verifying compiler F@BOOL@ for annotated computer programs, that uses effective and sound automatic SAT-solvers (i.e. programs that check satisfiability of propositional Boolean formulas in the conjunctive normal form) as means of automatic validation of correctness conditions (instead of semi-automatic proof techniques). The key idea is Boolean representation of all data instead of Boolean abstraction or first-order representation. (It makes difference between F@BOOL@ and SLAM.) Our project is aimed at verification of functional properties, and it assumes generation of first-order verification conditions (from invariants) and validation/refutation of each verification condition using SAT-solvers after their conservative translation into a Boolean form. During the period from 2006 to 2009, a popular (at that time) SAT-solver zChaff was used in the F@BOOL@ project. The first three verification experiments that have been exercised with its help are listed below: swapping values of two variables, checking whether three input values are lengths of sides of an equilateral or isosceles triangle, and detecting a unique fake in a set of 15 coins. The paper presents general outlines of the project and details of the last (the most extensive) experiment.

1. Introduction

A verifying compiler is a system computer program that translates programs written by a human ("user") from a high-level language into equivalent executable programs, and besides, proves (verifies) mathematical statements specified by the human about the properties of the programs being translated [10]. The purpose of the F@BOOL@ project is to develop a transparent for users, compact, portable and extensible verifying compiler F@BOOL@ for annotated computer programs, that uses effective and sound automatic SAT-solvers (i.e. programs that check satisfiability of propositional Boolean formulas in the conjunctive normal form) as means of automatic validation of correctness conditions (instead of semi-automatic proof techniques). The main target group of users of the F@BOOL@ compiler is the students of

*Partially supported by RFBR under Grant 09-01-00361-a.

mathematics, computer science, and information technology departments studying the basic combinatorics, sorting and search algorithms, basics of the formal methods (i.e. program specification and verification), etc. Since the computing programs transform their input data into the output ones, specifications of the computing programs are of the following two kinds. These kinds are the partial correctness conditions and the total correctness conditions. We are mostly interested in the partial correctness conditions. They are schematically written as $\{\phi\}\pi\{\psi\}$, where π is a program, ϕ is a precondition on the input data, and ψ is a postcondition on the output data. The partial correctness conditions are also known as Hoare triples [9]. A triple $\{\phi\}\pi\{\psi\}$ is said to be true (denoted by $\models \{\phi\}\pi\{\psi\}$) or the program π is said to be partially correct with respect to the precondition ϕ and the postcondition ψ, iff on any input data that satisfy the property ϕ the program π either does not stop (it loops forever, hangs up, etc.), or stops with output data that satisfy the property ψ [9]. An informal method (not an algorithm!) of determining the validity of partial correctness conditions has been developed in [8] and it became popular as the Floyd method for determining the validity of Hoare triples. Its correctness is well-known: if it is possible to apply it to a triple $\{\phi\}\pi\{\psi\}$, then $\models \{\phi\}\pi\{\psi\}$ [9].

The purpose of this paper is to present the current state of the project, in particular, by means of the most complicated verification example that has been exercised. This example is formal verification of a program that solves the following 15-coin puzzle [13]:

> • A set of 15 coins consists of 14 valid coins and a fake one. All valid coins have the same weight while the fake one has a different weight. The first coin is valid and is marked but all other coins (including the false one) are unmarked. Is it possible to identify the false coin using a balance at most 3 times?

Let us remark that verification of any program that solves the puzzle is not a "verification challenge", since it can be verified by a complete test suite that comprises 28 variants of the number and relative weight of the fake. But the purpose of this case study is to test the F@BOOL@ approach, not to solve the puzzle or a verification challenge.

In accordance with this purpose, the rest of the paper is organized as follows. Section 2 sketches the general outlines of the F@BOOL@ project and the current state of the art (as of September 1, 2010). Then Section 3 sketches the syntax and small-step operational semantics of a programming language mini-NIL(R, A), that is an extension of the kernel programming language of the project mini-NIL[1] by variable ranges (R) and static arrays (A). This section also sketches a transformational semantics of mini-NIL(R,

[1] NIL is acronym for "Non-deterministic Imperative Language".

A) that transforms mini-NIL(R, A) into the kernel language mini-NIL. The following Section 4 presents the 15-coin verification example: a method to detect the fake, a mini-NIL(R, A) annotated program implementation of the method, the result of transformation of the mini-NIL(R, A) program into an annotated mini-NIL program, and the final verification condition to be converted into SAT-format (while technical details are available in the Appendices). The paper is concluded by Section 5, where some directions for further research are presented.

2. F@BOOL@ at glance

Mini-NIL is a non-deterministic programming language similar to Basic, described in the project F@BOOL@ documentation [4, 5]. It consists of programs with preambles. The preamble defines the range of integer values and initializes variables. The programs are built of assignment and condition operators with non-deterministic control passing (transitions), labels, variables and constants (that are interpreted as elements of the additive ordered group of integer residuals modulo some maximal integer $2^n > 1$). At present the syntax of mini-NIL has a strict format, since the purpose of this language is not convenience and flexibility of programming, but "a proof of concept" of the F@BOOL@ project. The difference between the annotated and non-annotated programs in the mini-NIL language is that the preamble and some labels (including the initial and all final labels) have logical annotations associated with them. Informally speaking, the annotations are logical formulas constructed of equalities and inequalities over arithmetic expressions by means of usual logic operations of negation, conjunction, disjunction, implication, equivalence, and the universal and existence quantifiers. Informally speaking, annotations contributes to the program execution as "run-time contracts":

1. the precondition is checked on the input data (these data are specified in the preamble) and, in the case when this annotation appears incorrect, an exception "error in input data" is thrown;
2. the postcondition is checked on each set of the output results and, in the case when this summary appears incorrect, an exception "error in calculation results" is thrown;
3. before executing an operator marked by a label with an annotation, the annotation is checked on the current values of variables and, in the case when this annotation appears incorrect, an exception "run-time error" is thrown.

The static semantics of the annotated mini-NIL programs consists in construction of verification/correctness conditions. It is a concretization of

the Floyd method for partial correctness. Below, the static semantics of annotations is presented as an annotated pseudo-code.

Precondition. [A program π is a syntactically correct annotated mini-NIL program in which each operator has a unique label and, for each condition operator, its *then*-list and *else*-list are disjoint.]

1. Represent P as a flowchart with control points, so that
   (a) the start of the flowchart is a control point annotated by a precondition as an invariant;
   (b) any annotated label is a control point annotated by the corresponding invariant;
   (c) the end of the flowchart is a control point annotated by a postcondition as an invariant.
2. If there exists a loop through the flowchart that does not contain any control point, then the construction of the static semantics of annotations is immediately interrupted with an indefinite result; otherwise, it proceeds according to the next step.
3. For each control point l, construct (generate) the following correctness condition

$$\xi_l \to \Big(\bigwedge_{\substack{k \text{ is a control point,} \\ \xi_k \text{ is its invariant, and} \\ \pi_l^k \text{ is a loop-free path} \\ \text{from } l \text{ to } k}} WP(\pi_l^k, \xi_k) \Big),$$

   where ξ_l is the annotation (invariant) of the control point l, and WP is Dijkstra's weakest precondition transformer for loop-free programs [7].
4. The set of all generated correctness conditions is said to be the result of the construction of the static semantics for the annotated program π.

Postcondition. [For each initial state σ of a program π, if the precondition is valid in σ and all correctness conditions of π are tautologies, then the postcondition is valid in each final state that results from the initial σ.]

Soundness of the static semantics of an annotated mini-NIL program has been proved in technical report [5]. Therefore, by verification of mini-NIL programs we mean generation and validation of the correctness conditions of annotated programs. Let us remark that the above method for generation of correctness conditions is exponential in time and space because of

branching in programs and multiple variable instances in formulas. Therefore, in the framework of the F@BOOL@ project, a polynomial algorithm for correctness conditions generation has been developed and justified [12]. This algorithm uses auxiliary variables for invariants when generating the correctness conditions, and it can be applied both for non-structured non-deterministic programs and for structured deterministic programs. This algorithm linearly depends on the number of control constructs in a program and the number of statements, but has quadratic dependency on the total size of the program, precondition, postcondition and the invariants of the control points. This algorithm has been used in the 15-coin verification example manually (in section 4), but implementation of the algorithm is a future research topic.

The key ideas of F@BOOL@ are Boolean representation of all data (instead of Boolean abstraction or first-order representation) and the use of SAT-solvers for validation of the correctness conditions (instead of deductive reasoners). These ideas make difference between F@BOOL@ from one side and BLAST [6] and SLAM [3] verification tools from the other side. Both tools are static analyzers for a subset of the C language. They iteratively build and refine finite models of a program state-space by means of a so-called Boolean predicate abstraction, model-check program safety and liveness in these models by means of SAT-solvers and refute illegal program runs by means of first-order theorem-provers. In contrast, our project is aimed at verification of functional properties, and it assumes generation of first-order verification conditions (from invariants), and the validation/refutation of each verification condition using SAT-solvers after their "conservative" translation into the Boolean form by means of the following method.

Precondition. [θ is a first-order correctness condition over the additive ordered group of integer residuals modulo $2^n > 1$.]

1. $\xi := \mathit{bool}_n(\theta)$, where bool_n is an equivalent translation of first-order formulas over the additive ordered group of integer residuals modulo $2^n > 1$ into Boolean formulas;
2. $\chi := \mathit{cnf}_3(\neg\xi)$, where cnf_3 is an algorithm of translation of Boolean formulas into an equally satisfiable 3-cnf formula [1].

Postcondition. [A boolean formula χ is satisfiable iff the correctness condition θ is not a tautology.]

Let us note that step 2 of this algorithm has quadratic complexity on the size of the formula ξ, but the size of the resulting 3-cnf formula χ linearly depends on the size of ξ. However, step 1 has exponential complexity because of the replacement of universal quantifiers with conjunctions, and existential quantifiers with disjunctions. Therefore, at the current stage of implementation of the F@BOOL@ project, quantifiers in annotations are prohibited.

Provided this limitation, the complexity of step 1 becomes linear. During the period from 2006 to 2009, a popular at that time SAT-solver zChaff[2] was used in the F@BOOL@ project. The first verification experiments have been successfully made with its help. Our experience is bounded by the following toy Mini-NIL programs that

- swaps the values of two variables;
- checks whether three input values are the lengths of sides of an equilateral or isosceles triangle;
- solves the 15-coin puzzle.

3. Layers of NIL

The syntax, operational and transformational semantics of mini-NIL have been defined in [11] and are sketched below.

The syntax of mini-NIL(R, A) consists of programs. Every program consists of a preamble and a body. A program preamble is a list of variable and array declarations. A program body is a list of assignments to variables, updates of array elements and condition operators.

Program Preamble. A maximal integer declaration has the form '$MaxInt :: M$', where M is an unsigned integer constant greater than 1. A variable declaration has the form '$VAR\ x :\ [0..r]$', where x is an identifier (in low case letters), and r is an unsigned integer constant in the range $[0..M]$ (that is called a variable range). An array declaration has the form '$ARRAY\ a[r_1,\ ...r_n] :\ [0..r]$', where a is an identifier (in low case letters), and n is an unsigned positive integer constant, $r_1,\ ...r_n, r$ are unsigned integer constants in the range $[0..M]$; $r_1,\ ...r_n$ are called index ranges and r is called an element range. An identifier declaration is a variable or array declaration. A (program) preamble is a finite sequence of declarations that starts with a single maximal integer declaration and then consists of variable and array declarations such that every identifier has at most one declaration within this sequence.

Arithmetic expressions and array elements. Arithmetic expressions and array elements are defined by mutual induction as follows[3].

Arithmetic expressions:

- every unsigned integer in the range $[0..M]$ is a (simple) expression;
- every variable is a (simple) expression;
- every array element is a (compound) expression;

[2] `http://www.princeton.edu/~chaff/zchaff.html`

[3] We will use 'expression' and 'element' as a shorthand for an arithmetic expression and array element, respectively.

- every sum and difference of expressions is a (compound) expression;

Array element has the form '$a[\tau_1, ...\tau_n]$', where a is an array, n is a positive integer, and $\tau_1, ...\tau_n$ are arithmetic expressions (for element's indices).

Program body. A label is an unsigned integer 0, 1, 2, ... An assignment operator has the form '$l :\ x := \tau\ goto\ L$', where l is a label, x is a variable, τ is an arithmetic expression, and L is a finite sequence[4] of labels. An update operator has the form '$l :\ a[\tau_1, ...\tau_n] := \tau\ goto\ L$', where l is a label, $a[\tau_1, ...\tau_n]$ is an array element, τ is an arithmetic expression, and L is a finite sequence of labels. A condition operator has the form '$l :\ if\ \xi\ then\ L^+\ else\ L^-$', where l is a label, ξ is a quantifier-free formula constructed from equalities/inequalities of arithmetic expressions, L^+ and L^- are finite sequences[4] of labels. A (program) body is a finite set of operators[5] such that any label marks one operator at most. A label '0' (zero) is called an initial (or start) label. A final (or terminal) label of a body is any label that has an instance in the body but does not mark any operator[6].

Program. A preamble and a body are said to be consistent, if all variables and arrays used in the body are declared in the preamble, and all expressions in the body are type-correct with respect to the preamble. A program consists of a preamble followed by a body. If π is a program, then let us denote its preamble by $P(\pi)$ and its body by $B(\pi)$.

Thus, the syntax of a programming language Mini-NIL(R, A) is defined. It can be thought of as an extension of its kernel language mini-NIL [4] by static arrays and ranges for the values of variables, array indices and elements.

The operational semantics of mini-NIL(R, A) is called Small Step Semantics and expands the operational semantics of its kernel language. Informally speaking, execution of a mini-NIL(R, A) program starts from any operator marked by the label '0' and finishes with a pass of control to any label that does not mark any operator in the program. An exceptional situation occurs in execution when an indefinite value is assigned to a variable or when an indefinite array element is updated[7] or when control can not be passed to any definite label.

Small Step Semantics. A step (or small step) of a program π is a firing of any operator in π. A start configuration of π is any configuration with the label 0. A final configuration of π is any configuration with a label

[4]The empty sequence is admissible.

[5]I.e. the assignment, update, and condition operators

[6]It means that a terminal label occurs in '*goto*', '*then*', or '*else*' section(s) of some operator(s) but does not mark any operator in the body.

[7]But in contrast, update of a definite array element by an indefinite value does not rise an exception.

that does not mark any operator in π. A trace of π is any finite sequence of configurations such that every consequential pair of configurations within the sequence is a step of π. A computational trace of π is a trace that starts from a start configuration and finishes in a final configuration. The small step semantics of π is the following binary relation $SSS(\pi)$ on the state-space:

$$\{(\sigma' \ , \ \sigma'') \in \Sigma \times \Sigma \ : \ \text{there is a computational trace of } \pi \text{ that starts from the state } \sigma' \text{ and finishes in the state } \sigma''\}.$$

Let us observe that this definition is compatible with the definition of the operational semantics of the kernel language mini-NIL [4].

Paper [2] has suggested a 'split' of a computer language into a kernel layer, a number of intermediate layers and a complete layer. The kernel layer sublanguage should have a virtual machine semantics and provide tools for implementation of the intermediate layers; the intermediate layer sublanguages in turn should provide tools for the complete layer. Implementation of an intermediate layer sublanguage in the kernel layer sublanguage should be a semantics-preserving code transformation.

In the F@BOOL@ verification project, we would like to develop a verification-oriented programming language with mini-NIL as a kernel sublanguage and mini-NIL(R, A) as one of the intermediate layer sublanguages. It implies that we have to define some algorithm λ that transforms every mini-NIL(R, A) program π into a mini-NIL program $\lambda(\pi)$ such that the small step semantics $SSS(\pi)$ is 'equal' to the operational semantics $\{(\sigma' \ , \ \sigma'') \in \Sigma \times \Sigma \ : \ \sigma'' \in \lambda(\pi)(\sigma') \ \}$. The transformation λ comprises three steps [11]: first, program simplification, then array elimination, and finally, uniform ranging (i.e. shifting to the uniform range $[0..M]$).

Informally speaking, program simplification is a very intuitive procedure: replace any instance τ of a compound index or array element in a compound expression by a new variable y which should be 'initialized' by the value of τ in advance (i.e. $y := \tau$).

Intuition behind array elimination in a simple program is also very simple: just emulate any static array by two sets of new variables with indices for definite values and indefinite ones; for example, replace $ARRAY \ a[2] \ : \ [0..5]$ by three fresh variables $(VAR \ x0 \ : \ [0..5])$, $(VAR \ x1 \ : \ [0..5])$, $(VAR \ x2 \ : \ [0..5])$ for representing the values of $a[0]$, $a[1]$ and $a[2]$ when they are definite, and three fresh variables $(VAR \ y0 \ : \ [0..1])$, $(VAR \ y1 \ : \ [0..1])$, $(VAR \ y2 \ : \ [0..1])$ for indicating whether the values of $a[0]$, $a[1]$ and $a[2]$ are indefinite.

The idea behind range uniformation is very trivial: if the range r of a variable x is not equal to $MaxInt$, then, before any assignment to this variable $x := \tau$, test whether the value of τ is in the range $[0..r]$.

4. Case study: 15-coin puzzle

The 15-coin puzzle is a hard problem[8]. Nevertheless, let us present below how to solve it in a human-friendly notation.

Let us divide the coins into three sets: the first set comprises the marked valid coin and 4 unmarked coins, the second and third sets comprise unmarked coins (five coins in each). Then let us balance the first set against the second one. We can have three outcomes after the first balancing: $(<)$ the first set is lighter, $(>)$ the second set is lighter, and $(=)$ they are equal.

First let us discuss the last outcome $(=)$. In this case the fake is in the third set, while all coins in the first and second sets are valid. In this case let us balance any coin x in the third set with any valid coin against any two other coins y and z in the third set. Again, we can get three outcomes: $(=,<)$ the first pair is lighter, $(=,>)$ the second pair is lighter, and $(=,=)$ the pairs are equal. If pairs are equal $(=,=)$ then the fake is one of two non-tested coins in the third set, and it can be detected by balancing any of them against a valid coin. In the subcase $(=,<)$ the fake is either x, y or z; it can be detected by balancing x and y against any pair of valid coins: if $x\&y$ are lighter than valid coins, then x is the fake, if they are heavier than the valid, then the fake is y, otherwise the fake is z. The subcase $(=,>)$ is similar to the previous one. It finishes the case $(=)$.

Let us discuss the first outcome $(<)$. In this case, the fake is one of 9 unmarked coins in two sets, while all 5 coins in the third set are valid. Let us separate any unmarked coin x from the first set and any unmarked coins y and z from the second set, and then balance the remaining 6 unmarked coins in the first and second sets against 6 valid coins (those are the marked coin and 5 coins from the third set). Again, three outcomes are possible: $(<,<)$ unmarked coins are lighter, $(<,>)$ they are heavier, and $(<,=)$ they are equal to valid coins. In the subcase $(<,=)$, the fake is x, y or z, but we already know that the marked coin with x are lighter than $y\&z$; hence the fake can be detected by a single balancing similarly to the previous subcase $(=,<)$. In the subcase $(<,<)$, we know that the fake is lighter and a coin among three unmarked from the first set; a single light in three coins can be detected by balancing any two of them once. The subcase $(<,>)$ is similar to the previous subcase $(<,<)$, but the fake is heavier in this case. It finishes the case $(<)$.

The last case $(>)$ is similar to $(<)$.

An annotated pseudo-code that formalizes the above method is presented in the Appendix A. The corresponding annotated mini-NIL(R, A) code is presented in the Appendix B. An 'equivalent' annotated mini-NIL program is presented in the Appendix C. This program has been constructed

[8]Please refer to the expository paper [13] for a "story" how a parameterized variant of the puzzle can be solved with the help of finite games and fix-point calculus.

manually according to transformational semantics [11], but with a simpler array elimination (that is correct, since all elements of the array M[0..14] are defined as it follows from the precondition). The correctness condition θ (generated manually according to [12]) is presented in the Appendix D. Then (according to F@BOOL@ outlines sketched in section 2) the following boolean formula $cnf_3(\neg(bool_4(\theta)))$ has been constructed automatically (where $bool_n$ is an equivalent translation of first-order formulas over the additive ordered group of integer residuals modulo $2^4 = 16 > 14$ into Boolean formulas, and cnf_3 is an algorithm of translation of Boolean formulas into an equally satisfiable 3-cnf formula [1]) and refuted by SAT-solver zChaff[9].

5. Concluding remarks: what's next?

In this paper, we gave a brief overview of the F@BOOL@ project, presented an intermediate layer of the project programming language mini-NIL with variable ranges and static arrays, and the most complicated verification example that has been exercised so far. The intermediate layer programming language has been provided with operational and transformational semantics. Below we present and motivate some future research directions.

1. Implement the transformational semantics of mini-NIL [11].
2. Implement the generator of verification condition that has been described in [12].
3. Try F@BOOL@ for verification of more interesting examples than before.
4. Complete the manual proof-sketches from [11] with any proof-assistance.

We think that the first three research topics are quite natural and we can skip any motivation for them. In contrast, we would like to justify the last research topic by the following quotation from the Call For Papers[10] of the 4rd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory:

> Researchers in programming languages have long felt the need for tools to help formalize and check their work. With advances in language technology demanding deep understanding of ever larger and more complex languages, this need has become urgent. There are a number of automated proof assistants being developed within the theorem proving community that seem ready or nearly ready to be applied in this domain. Yet, despite numerous individual efforts in this direction, the use of proof

[9] `http://www.princeton.edu/~chaff/zchaff.html`
[10] `http://www.seas.upenn.edu/~sweirich/wmm/`

assistants in programming language research is still not commonplace: the available tools are confusingly diverse, difficult to learn, inadequately documented, and lacking in specific library facilities required for work in programming languages.

References

[1] Aho A.V., Hopcroft J.E., Ullmann J.D. The design and analysis of computer algorithms. – Addison-Wesley, 1974.

[2] Anureev I.S., Bodin E.V., Gorodnyaya L.V., Marchuk A.G., Murzin F.A., Shilov N.V. On the problem of computer language classification // Joint NCC&IIS Bull. Ser. Computer Science. – 2008. – Iss. 28. – P. 1–29.

[3] Ball T., Cook B., Levin V., and Rajamani S. K. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft// Lect. Notes Comput. Sci. – 2004. – Vol. 2999. – P. 1–20.

[4] Bodin E., Kalinina N., Shilov N. Verifying Compiler F@BOOL@ Part I: Outlines of F@BOOL@ project in the context of component-based programming. Mini-NIL: a prototype of F@BOOL@ virtual machine language. – Novosibirsk, 2005. – (Prepr. / IIS SB RAS; N 131).

[5] Bodin E., Kalinina N., Shilov N. Verifying Compiler F@BOOL@ Part II: Logical annotations in mini-NIL, their static and run-time semantics. – Novosibirsk, 2006. – (Prepr. / IIS SB RAS; N 138).

[6] Beyer D., Henzinger T.A., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering// Int. J. on Software Tools for Technology Transfer. – 2007. – No. 9. – P. 505–525.

[7] Dijkstra W.E. The Dicsipline of programming. – Prentice Hall, 1976.

[8] Floyd R.W. Assigning meanings to programs // Proc. of a Symposium in Applied Mathematics. Mathematical Aspects of Computer Science. – American Math. Society, Providence, R. I., 1967. – Vol. 19. – P. 19–32.

[9] Gries D. The Science of Programming. – NY: Springer Verlag, 1981.

[10] Hoare C. A. R. The verifying compiler: a grand challenge for computing research // Perspectives of Systems Informatics (PSI'2003). – Lect. Notes Comput. Sci. – 2003. – Vol. 2890. – P. 1–12.

[11] Shilov N.V., Bodin E.V., Shilova S.O. Fabulous arrays I: Operational and transformational semantics of static arrays in verification project F@BOOL@ // Bull. Nov. Comp. Center. Ser. Comp. Science. – Novosibirsk, 2009. – IIS Special Iss. 29. – P. 121–140.

[12] Shilov N.V., Anureev I.S., and Bodin E.V. Generation of verification conditions for imperative programs // Programming and Computer Software. – 2008. – Vol. 34, N 6. – P. 307–321.

[13] Shilov N.V., Yi K. How to find a coin: propositional program logics made easy // Current Trends in Theoretical Computer Science, World Scientific. – 2004. – Vol. 2. – P. 181–214.

A. Annotated pseudo-code

[There are 15 coins M0, ... M14,
& 13 coins in M1, ... M14 have weight that is equal to M0,
& a single coin in M1, ... M14 has another weight.]

If (weight of) M0 + M1 ... + M4 equals (weight of) M5 + M6 + ... M9
then

1. if M0 + M10 equals M11 + M12
   (a) then {if M0 equals M13 then (fake number) F:= 14 else F:= 13}
   (b) else {if M0 + M1 equals M10 + M11 then F:= 12 else {if the first weight was greater than the second in balances 1 and 1(b) then F:= 11 else F:= 10}}

else

2. if M2 + M3 + M4 + M5 + M6 + M7 is not equal to M0 + M10 + M11 ... + M14
   then
   (a) if the first weight was less than the second in balances 2
   then
      i. if M2 equals M3 then F:= 4 else {if the first weight was less than the second in balance 2(a)i then F:= 2 else F:= 3}
   else
      ii. if M5 equals M6 then F:= 7 else {if the first weight was greater than the second in balance 2(a)ii then F:= 5 else F:= 6}
   else
   (b) if the first weight was less than the second in the initial balance
   then
      i. if M0 + M10 equals M1 + M8 then F:= 9 else {if the first weight was less than the second in balance 2(b)i then F:= 8 else F:= 1}
   else
      ii. if M0 + M10 equals M1 + M8 then F:= 9 else {if the first weight was less than the second in balance 2(b)ii then F:= 1 else F:= 8}

[The coin with the number F has weight that differs from that of M0.]

B. Annotated mini-NIL(R, A)

```
MaxInt ::  14 ; // Maximal Integer.
VAR F : [0..14] ; // Program variable.
ARRAY M[14] :  [0..2] ; // Program array.
VAR I : [0..14] ; VAR J : [0..14] ; // Quantifier variables.
(M[0] = 1 & E I.((M[I] = 0 V M[I] = 2) & A J.(J <> I => M[J] = 1)))
                                                    // Precondition.
:  // Program body begin.
0:  if M[0] + M[1] + M[2] + M[3] + M[4] =
                 M[5] + M[6] + M[7] + M[8] + M[9] then {1} else {10}
1:  if M[0] + M[10] = M[11] + M[12] then {2} else {5}
2:  if M[0] = M[13] then {3} else {4}
3:  F:= 14 goto {33}
4:  F:= 13 goto {33}
5:  if M[0] + M[1] = M[10] + M[11] then {6} else {7}
6:  F:= 12 goto {33}
7:  if M[0] + M[10] > M[11] + M[12] & M[0] + M[1] > M[10] + M[11]
                                                  then {8} else {9}
8:  F:= 11 goto {33}
9:  F:= 10 goto {33}
10:  if M[2] + M[3] + M[4] + M[5] + M[6] + M[7] ≠
       M[0] + M[10] + M[11] + M[12] + M[13] + M14 then {11} else {22}
11:  if M[2] + M[3] + M[4] + M[5] + M[6] + M[7] <
       M[0] + M[10] + M[11] + M[12] + M[13] + M14 then {12} else {17}
12:  if M[2] = M[3] then {13} else {14}
13:  F:= 4 goto {33}
14:  if M[2] < M[3] then {15} else {16}
15:  F:= 2 goto {33}
16:  F:= 3 goto {33}
17:  if M[5] = M[6] then {18} else {19}
18:  F:= 7 goto {33}
19:  if M[5] > M[6] then {20} else {21}
20:  F:= 5 goto {33}
21:  F:= 6 goto {33}
22:  if M[0] + M[1] + M[2] + M[3] + M[4] <
                M[5] + M[6] + M[7] + M[8] + M[9] then {23} else {28}
23:  if M0 + M10 = M1 + M8 then {24} else {25}
24:  F:= 9 goto {33}
25:  if M0 + M10 < M1 + M8 then {26} else {27}
26:  F:= 8 goto {33}
27:  F:= 1 goto {33}
28:  if M[0] + M[10] = M[1] + M[8] then {29} else {30}
```

```
29:  F:= 9 goto {33}
30:  if M[0] + M[10] < M[1] + M[8] then {31} else {32}
31:  F:= 1 goto {33}
32:  F:= 8 goto {33}
:  // Program body end.
M[F] ≠ 1.  // Postcondition.
```

C. Annotated mini-NIL code

```
14 ; // Maximal Integer.
0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 0; 0;
    // ‘‘Random’’ initial values for F, M0, ...  M14, I, J that meet
precondition.
(M0 = 1 & (
(M0 = 0 V M0 =2 ) & ((0 ≠ 0 => M0 = 1) & (1 ≠ 0 => M1 = 1) & ...
                                          & (14 ≠ 0 => M14 = 1)) V
(M1 = 0 V M1 =2 ) & ((0 ≠ 1 => M0 = 1) & (1 ≠ 1 => M1 = 1) & ...
                                          & (14 ≠ 1 => M14 = 1)) V
.............................................................
(M14 = 0 V M14 =2 ) & ((0 ≠ 14 => M0 = 1) &
               (1 ≠ 14 => M1 = 1) & ...  & (14 ≠ 14 => M14 = 1)) ))
                                                  // Precondition.
:  // Program body begin.
0:  if M0 + M1 + M2 + M3 + M4 = M5 + M6 + M7 + M8 + M9
                                                 then {1} else {10}
1:  if M0 + M10 = M11 + M12 then {2} else {5}
2:  if M0 = M13 then {3} else {4}
3:  F:= 14 goto {33}
4:  F:= 13 goto {33}
5:  if M0 + M1 = M10 + M11 then {6} else {7}
6:  F:= 12 goto {33}
7:  if M0 + M10 > M11 + M12 & M0 + M1 > M10 + M11 then {8} else {9}
8:  F:= 11 goto {33}
9:  F:= 10 goto {33}
10:  if M2 + M3 + M4 + M5 + M6 + M7 ≠
                M0 + M10 + M11 + M12 + M13 + M14 then {11} else {22}
11:  if M2 + M3 + M4 + M5 + M6 + M7 <
                M0 + M10 + M11 + M12 + M13 + M14 then {12} else {17}
12:  if M2 = M3 then {13} else {14}
13:  F:= 4 goto {33}
14:  if M2 < M3 then {15} else {16}
15:  F:= 2 goto {33}
16:  F:= 3 goto {33}
17:  if M5 = M6 then {18} else {19}
18:  F:= 7 goto {33}
19:  if M5 > M6 then {20} else {21}
```

```
20:  F:= 5 goto {33}
21:  F:= 6 goto {33}
22:  if M0 + M1 + M2 + M3 + M4 < M5 + M6 + M7 + M8 + M9
                                                  then {23} else {28}
23:  if M0 + M10 = M1 + M8 then {24} else {25}
24:  F:= 9 goto {33}
25:  if M0 + M10 < M1 + M8 then {26} else {27}
26:  F:= 8 goto {33}
27:  F:= 1 goto {33}
28:  if M0 + M10 = M1 + M8 then {29} else {30}
29:  F:= 9 goto {33}
30:  if M0 + M10 < M1 + M8 then {31} else {32}
31:  F:= 1 goto {33}
32:  F:= 8 goto {33}
:  // Program body end.
( (F = 0 => M0 ≠ 1) & (F = 1 => M1 ≠ 1) & ...
                                          (F = 14 => M14 ≠ 1) ).
                                               // Postcondition.
```

D. Correctness condition θ

```
Conjunction of the following formulas ini, 0--32 and fin.
ini:  precondition from Appendix C => P0
0:  ( (M0 + M1 + M2 + M3 + M4 = M5 + M6 + M7 + M8 + M9) <=> Q0 ) =>
                                     ( P0 => (Q0 & P1) V (∼Q0 & P10) )
1:  ( (M0 + M10 = M11 + M12) <=> Q1 ) =>
                                      ( P1 => (Q1 & P2) V (∼Q1 & P5) )
2:  ( (M0 = M13) <=> Q2 ) => ( P2 => (Q2 & P3) V (∼Q2 & P4) )
3:  P3 => (P33 & F=14)
4:  P4 => (P33 & F=13)
5:  ( (M0 + M1 = M10 + M11) <=> Q5 ) =>
                                      ( P5 => (Q5 & P6) V (∼Q5 & P7) )
6:  P6 => (P33 & F=12)
7:  ( (M0 + M10 > M11 + M12 & M0 + M1 > M10 + M11) <=> Q7 ) =>
                                      ( P7 => (Q7 & P8) V (∼Q7 & P9) )
8:  P8 => (P33 & F=11)
9:  P9 => (P33 & F=10)
10:  ((M2 + M3 + M4 + M5 + M6 + M7≠M0 + M10 + M11 + M12 + M13 + M14)
                    <=> Q10 )=>( P10 => (Q10 & P11) V (∼Q10 & P22) )
11:  ((M2 + M3 + M4 + M5 + M6 + M7 < M0 + M10 + M11 + M12 + M13 + M14)
                   <=> Q11 ) => ( P11 => (Q11 & P12) V (∼Q11 & P17) )
12:  ( (M2 = M3) <=> Q12 ) => ( P12 => (Q12 & P13) V (∼Q12 & P14) )
13:  P13 => (P33 & F=4)
14:  ( (M2 < M3) <=> Q14 ) => ( P14 => (Q14 & P15) V (∼Q14 & P16) )
15:  P15 => (P33 & F=2)
16:  P16 => (P33 & F=3)
17:  ( (M5 = M6) <=> Q17 ) => ( P17 => (Q17 & P18) V (∼Q17 & P19) )
```

```
18:  P18 => (P33 & F=7)
19:  ( (M5 > M6) <=> Q19 ) => ( P19 => (Q19 & P20) V (∼Q19 & P21) )
20:  P20 => (P33 & F=5)
21:  P21 => (P33 & F=6)
22:  ((M0 + M1 + M2 + M3 + M4 < M5 + M6 + M7 + M8 + M9) <=> Q22 ) =>
                                ( P22 => (Q22 & P23) V (∼Q22 & P28) )
23:  ( (M0 + M10 = M1 + M8) <=> Q23 ) =>
                                ( P23 => (Q23 & P24) V (∼Q23 & P25) )
24:  P24 => (P33 & F=9)
25:  ( (M0 + M10 < M1 + M8) <=> Q25 ) =>
                                ( P25 => (Q25 & P26) V (∼Q25 & P27) )
26:  P26 => (P33 & F=8)
27:  P27 => (P33 & F=1)
28:  ( (M0 + M10 = M1 + M8) <=> Q28 ) =>
                                ( P28 => (Q28 & P29) V (∼Q28 & P30) )
29:  P29 => (P33 & F=9)
30:  ( (M0 + M10 < M1 + M8) <=> Q30 ) =>
                                ( P30 => (Q30 & P31) V (∼Q30 & P32) )
31:  P31 => (P33 & F=1)
32:  P32 => (P33 & F=8)
fin:  P33 => postcondition from Appendix C
```