Bull. Nov. Comp. Center, Comp. Science, 29 (2009), 119–138

Fabulous arrays I: Operational and transformational semantics of static arrays in verification project F@BOOL@*

N. V. Shilov, Eu. V. Bodin, S. O. Shilova

To memory of our colleague Nina Kalinina.

Abstract. 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 programs for checking satisfiability of propositional Boolean formulas. The kernel programming language of the project interprets all variables by residuals modulo some fixed integer (that is a parameter). Paper presents an extension of the kernel language by different variable ranges and static multi-dimensional arrays, provides two kinds of semantics of the extension — operational and transformational (into the kernel language), sketches manual correctness proof for the transformational semantics.

1. Introduction

A verifying compiler is a system computer program that translates programs written by a human 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 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) and formal description techniques (for distributed systems protocols). But we also hope that F@BOOL@ can be applied in practice for specification, development and verification of device drivers as well. 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

*The work is supported by RFBR grant 09-01-00361-a.

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 Hoare 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 of 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 the present paper is to present syntax, operational and transformational semantics of an extension of the kernel programming language mini-NIL by variables with different ranges and by many-dimensional static arrays and to prove manually correctness of the transformational semantics with respect to operational one. In accordance with this purpose, the rest of the paper is organized as follows. Section 2 sketches the general outlines of F@BOOL@ project and current state of the art (as of November 1, 2009). Then Section 3 formally defines the syntax of mini-NIL(R, A) — an extension of mini-NIL by variable ranges and static arrays. A small step operational semantics for mini-NIL(R, A) is defined in Section 4. The transformational semantics of mini-NIL(R, A) that transforms mini-NIL(R, A) into the kernel language mini-NIL is defined in Section 5 together with a manual proof of its correctness. The paper is concluded by Section 6, 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 "to proof the 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 initial and all final labels) have logical annota-

tions 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 calculations 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 of mini-NIL. Below, the static semantics of annotations is presented as annotated pseudo-code.
Precondition. [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 any loop through the flowchart does not contain any control point, then the construction of the static semantics of the 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

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

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 [11]. This algorithm uses auxiliary variables for invariants when generating the correctness conditions, and it can be applied both for non-structured nondeterministic programs and for structured deterministic programs. This algorithm linearly depends on the number of the control constructs in the 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. 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 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 limited 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 on the verification of a wide spectrum of functional and behavioral

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 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. $\xi := bool_n(\theta)$, 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. $chi := cnf_3(\neg\xi)$, where cnf_3 is an algorithm of translation of Boolean formulas into an equally satisfiable 3-cnf formula [1].

Postcondition. [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 annotation are prohibited. Provided this limitation, the complexity of step 1 becomes linear. During the period from 2006 to 2008, a popular at that time SAT-solver zChaff[1] 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 values of two variables;
- checks whether three input values are lengths of sides a triangle;
- finds a unique fake coin in a set of 15 coins.

3. Syntax of Mini-NIL with ranges and arrays

Syntax of Mini-NIL with ranges and arrays 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

[1] http://www.princeton.edu/ chaff/zchaff.html

'*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. If δ is a preamble, then we denote the set of declared variables by $V(\delta)$ and the set of declared arrays by $A(\delta)$ (δ may be omitted when it is implicit).

Arithmetic expressions and array elements. Arithmetic expressions and array elements are defined by mutual induction as follows[2].

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;
- 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 indexes).

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[3] 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[3] of labels. A (program) body is a finite set of operators[4] 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[5]. If β is a body, then let us denote the set of its labels by $L(\beta)$ and the set of its final labels by $F(\beta)$ (β may be omitted when it is implicit).

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

[3]The empty set is admissible.

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

[5]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.

Program. A preamble and a body are said to be consistent, if all variables and arrays that are 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 syntax of a programming language is defined. We will call this language mini-NIL with ranges and arrays and denote by Mini-NIL(R, A), where 'R' stands for 'Ranges' and 'A' stands for 'Arrays'. It can be thought of as an extension of its kernel language mini-NIL [4] by static arrays and ranges for values of variables, array indexes and elements[6].

Simple programs. A special class of mini-NIL(R, A) programs comprises so-called simple programs that have no array elements as arguments of compound arithmetic expressions and no compound arithmetic expressions as indexes of array elements.

4. Small step semantics of mini-NIL(R, A)

Operational semantics of Mini-NIL with ranges and arrays is called Small Step Semantics and expands operational semantics of its kernel language. Informally speaking, execution of a mini-NIL(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.

Let us fix for awhile a program π. It consists of a preamble $\delta \equiv P(\pi)$ and a body $\beta \equiv B(\pi)$.

Semantics of preamble states. A state σ is a mapping that assigns integers to all declared variables and partial integer functions to all declared arrays as follows:

- if '$VAR\, x :\ [0..r]$;' is a variable declaration within δ, then $\sigma(x) \in [0..r]$;
- if '$ARRAY\ a[r_1,\ ...r_n] :\ [0..r]$;' is an array declaration within δ, then $\sigma(a) : [0..r_1] \times \ldots [0..r_n] \ \rightarrow\ [0..r]$ is a partial function.

The set of all states is denoted by $\Sigma(\pi)$ and called a state-space (of π).

Values of expressions and elements in a state. Every state σ assigns some (definite or indefinite) values to arithmetic expressions and array elements:

Values of expressions:

[6] In mini-NIL, the range of all variables is uniform $[0..M]$ and arrays are not permitted.

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

- if an expression τ is some unsigned integer $n \in [0..M]$, then the value $\sigma(\tau)$ is n;
- if an expression τ is some declared variable x, then the value $\sigma(\tau)$ is $\sigma(x)$;
- if an expression τ is an element $a[\tau_1, \dots \tau_n]$ of some declared array, then the value $\sigma(\tau)$ is the value of the element $\sigma(a[\tau_1, \dots \tau_n])$;
- if an expression τ is the sum/difference of expressions τ_1 and τ_2, and the values $\sigma(\tau_1)$ and $\sigma(\tau_2)$ are definite, then the value $\sigma(\tau)$ is $(\sigma(\tau_1)+\sigma(\tau_2))mod(M+1)$ or $(\sigma(\tau_1)-\sigma(\tau_2))mod(M+1)$, respectively;
- otherwise, the value $\sigma(\tau)$ is indefinite.

Value of element: if all values $\sigma(\tau_1)$, ... $\sigma(\tau_n)$ are definite and equal to some integers t_1, ... t_n, respectively, and the function $\sigma(a)$ is definite at the point $(t_1, \dots t_n)$, then the value $\sigma(a[\tau_1, ...\tau_n])$ is $\sigma(t_1, \dots t_n)$; otherwise, the value $\sigma(a[\tau_1, ...\tau_n])$ is indefinite.

Since we have to compare definite and indefinite values, we have to adopt 3-value logic (instead of Boolean logic) with logical values $\{true,\ false,\ absurd\}$ (that we abbreviate to 't', 'f' and 'a', respectively).

3-value logic in a state. Let σ be a state. The logical value in the state $\sigma(\phi)$ of a quantifier-free first-order formula ϕ is defined by induction on the structure of ϕ as follows.

- Equalities: $\sigma(\tau' = \tau'') = \begin{cases} true, \text{ if both values } \sigma(\tau'),\ \sigma(\tau'') \\ \qquad\qquad \text{are defined and equal,} \\ \qquad\qquad \text{or both are indefinite;} \\ false, \text{ if both values } \sigma(\tau'),\ \sigma(\tau'') \\ \qquad\qquad \text{are defined but not equal,} \\ \qquad\qquad \text{or if one of them is definite} \\ \qquad\qquad \text{while another is indefinite;} \end{cases}$.
- Inequalities:
$$\sigma(\tau' < (\leq,\ dots)\tau'') = \begin{cases} true, \text{ if both values } \sigma(\tau'),\ \sigma(\tau'') \\ \quad \text{are defined and the first is less than} \\ \quad \text{(less than or equal to, ...) the second;} \\ false, \text{ if both values } \sigma(\tau'),\ \sigma(\tau'') \\ \quad \text{are defined and the first is not less than} \\ \quad \text{(not less than or equal to, ...) the second;} \\ absurd, \text{ otherwise;} \end{cases}.$$
- The values for propositional combinations of equalities and inequalities are computed from the values of components in accordance with the following truth-tables:

arg_1	arg_2	$\wedge$	$\vee$
t	t	t	t
t	f	f	t
t	a	a	a
f	t	f	t
f	f	f	f
f	a	a	a
a	t	a	a
a	f	a	a
a	a	a	a

arg	$\neg$
t	f
f	t
a	a

Firing. A normal configuration is a pair of the form (l, σ), where l is a label and σ is a state. Abnormal or exceptional configurations are $IndVal$, $CtrLos$, and $CtrLosIndVal$, where '$IndVal$' stands for 'Indefinite Value', and '$CtrLos$' — for 'Control Loss'. Firing of an operator is a pair of configurations defined below.

Assignment firing. Let '$l: \ x := \tau \ goto \ L$' be an assignment operator, where a variable is declared by '$VAR \ x : \ [0..r];$'. A normal firing of the assignment is a pair of configurations $((l, \sigma), (l', \ \sigma'))$ such that $l' \in L$, the value $\sigma(\tau)$ is equal to some $t \in [0..r]$, and $\sigma' = upd(\sigma, x, t)$. An abnormal firing of the assignment is a pair of configurations of one of the following three forms:

- $((l, \sigma), \ IndVal)$ if $\sigma(\tau)$ is indefinite or is out of the range $[0..r]$, but $L \neq \emptyset$;
- $((l, \sigma), \ CtrLos)$ if $\sigma(\tau)$ is definite and is in the range $[0..r]$, but $L = \emptyset$,
- $((l, \sigma), \ CtrLosIndVal)$, if $\sigma(\tau)$ is indefinite or is out of the range $[0..r]$ and $L = \emptyset$.

Update firing. Let '$l: \ a[\tau_1, \ldots \tau_n] := \tau \ goto \ L$' be an update operator, where an array is declared by '$ARRAY \ a[r_1, \ldots r_n] : \ [0..r];$'. A normal firing of the update is a pair of configurations $((l, \sigma), (l', \ \sigma'))$ such that $l' \in L$, the values of $\sigma(\tau_1)$, ... $\sigma(\tau_n)$ are definite and are equal to some values $t_1 \in [0..r_1]$, ... $t_n \in [0..r_n]$, respectively, and $\sigma' = upd(\sigma, a, upd(\sigma(a), (t_1, \ldots t_n), t))$, where t is either the value $\sigma(\tau)$, if it is definite and is in the range $[0..r]$, or is indefinite value otherwise. An abnormal firing of the update is a pair of configurations of one of the following three forms:

- $((l, \sigma), \ IndVal)$ if any $\sigma(\tau_i)$, $i \in [1..n]$, is indefinite or is out of the range $[0..r_i]$, but $L \neq \emptyset$;

- $((l, \sigma),\ CtrLos)$ if the values of $\sigma(\tau_1)$, ... $\sigma(\tau_n)$ are definite and are in the ranges $[0..r_1]$, ... $[0..r_n]$, respectively, but $L = \emptyset$,
- $((l, \sigma),\ CtrLosIndVal)$, if any $\sigma(\tau_i)$, $i \in [1..n]$, is indefinite or is out of the range $[0..r_i]$ and $L = \emptyset$.

Condition firing. Let '$l :\ if\ \xi\ then\ L^+\ else\ L^-$' be a condition operator. A normal firing of the assignment is a pair of configurations $((l, \sigma), (l',\ \sigma))$ such that

- either $\sigma(\xi) = true$ and $l' \in L^+$,
- or $\sigma(\xi) = false$ and $l' \in L^-$.

An abnormal firing of the condition operator is a pair of configurations of one of the following three forms:

- $((l, \sigma),\ IndVal)$ if $\sigma(\xi) = absurd$;
- $((l, \sigma),\ CtrLos)$ if $\sigma(\xi) = true$ and $L^+ = \emptyset$;
- $((l, \sigma),\ CtrLos)$ if $\sigma(\xi) = false$ and $L^- = \emptyset$.

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 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. Small step semantics of π is the following binary relation $SSS(\pi)$ on the state-space:

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

Thus the operational semantics of the programming language Mini-NIL with ranges and arrays Mini-NIL(R, A) is defined. Let us observe that this definition is compatible with the definition of the operational semantics of the kernel language mini-NIL [4] in the following sense.

Proposition 1. *Assume that ρ is a mini-NIL program with M as the maximal integer. Let ρ_R be a mini-NIL(R,A) program that results from ρ by adding to the preamble the variable declaration $VAR\ x\ :\ [0..M]$ for every variable x that occurs in ρ. Then the small step semantics $SSS(\rho_R)$ is equal to $\{(\sigma'\ ,\ \sigma'') \in \Sigma \times \Sigma\ :\ \sigma'' \in \rho(\sigma')\ \}$, where $\rho(\dots)$ is the input-output operational semantics of ρ [4].*

Proof. Let us remark that the mini-NIL(R,A) program ρ_R is array free; hence, in every state σ, the value of every expression $\sigma(\tau)$ is definite. Due to this reason, *absurd* can not be the value of any propositional combination of equalities and inequalities in any state; hence all firings of all condition operators are either normal firings or abnormal firings of the form $((l, \sigma),\ CtrLos)$ (where l is a label and σ is a state), that are defined exactly as normal and abnormal firings of condition operators in mini-NIL [4]. Since all variables have a unique range $[0..M]$ and the value of all expressions in all states are definite values in the same range, all firings of all assignment operators are either normal firings or abnormal firings of the form $((l, \sigma),\ CtrLos)$ (where l is a label and σ is a state), that are defined exactly like normal and abnormal firings of assignment operators in mini-NIL [4]. It implies that definitions of the computational trace for programs ρ and ρ_R are equivalent. Since for every state σ', the set of states $\rho(\sigma')$ has been defined in [4] as $\{\sigma'') \in \Sigma$: there is a computational trace of π that starts from the state σ' and finishes in the state $\sigma''\}$, $SSS(\rho_R) = \{(\sigma'\ ,\ \sigma'') \in \Sigma \times \Sigma\ :\ \sigma'' \in \rho(\sigma')\ \}$. ■

5. Transformational semantics

Paper [2] has suggested '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 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 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 will consist of three steps: first program simplification, then array elimination, and finally uniform ranging (i.e. shifting to the uniform range $[0..M]$).

5.1. Program simplification

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. Let us present an annotated pseudo-code of an algorithm that we will refer to as program simplification.

Precondition: [π is a mini-NIL(R, A) program, δ is its preamble, and S is its small-step semantics.]
Algorithm:
WHILE π is not simple DO
BEGIN
LET $(l:\ op)$ be an operator within π that contains
(1) either an array element in a compound arithmetic expression,
(2) or a compound expression as an array index;

LET τ be an instance of
- either an array element in a compound expression,
- or a compound expression that is an array index within this operator;

LET r be
- either the range of the corresponding array element if τ is an instance of an array element,
- or the corresponding index range if τ is an instance of a compound expression as array index;

LET y be a fresh variable (i.e. a new identifier),
and LET k be a fresh label (i.e. an unsigned integer);

π := ADD variable declaration $(VAR\ y\ :\ [0..r];)$ to the preamble of π, and
REPLACE operator $(l:\ op)$ in the body of π by two operators $(l:\ y := \tau\ goto\ \{k\})$ and $(k:\ op_{y/\tau})$, where $op_{y/\tau}$ is instantiation of y instead of τ
END.
Postcondition:
[π is a simple mini-NIL(R, A) program, and its small step semantics $SSS(\pi)$ restricted onto variables and arrays declared in δ is equal to S.]

Proposition 2. *The program simplification algorithm is totally correct with respect to its precondition and postcondition.*

Proof (sketch). Let us proof partial correctness first and termination then. In both cases, we will use proof techniques developed by R. Floyd [8, 9], namely, the loop invariant and the potential function.

To prove partial correctness of the algorithm, let us adopt the following 'mix' of the precondition and postcondition

π is a mini-NIL(R, A) program, and
$SSS(\pi)$ restricted onto variables and arrays declared in δ is equal to S

as the invariant of the single loop of the algorithm. It is straightforward that

- the precondition implies the invariant,
- the invariant and negation of the loop condition imply the post condition.

It remains to prove that if the loop invariant holds before a legal iteration of the loop body and the iteration terminates, then it also holds after the iteration. It is sufficient to observe that a pair of configurations $((l, \sigma),\ (l', \sigma'))$ is a firing of the operator $(l :\ op)$ iff $((l, \sigma),\ (k,\ upd(\sigma, y, \sigma(\tau))))$ is a firing of $(l :\ y := \tau\ goto\ \{k\})$ and $((k,\ upd(\sigma, y, \sigma(\tau)),\ (l', \sigma'))$ is a firing of the operator $(k :\ op_{y/\tau})$, where $op_{y/\tau}$ is instantiation of y instead of τ. Thus partial correctness is proved.

Termination can be proved by adopting the following mapping

$$F :\ \pi\ \mapsto \left(\begin{array}{c}\text{the total number in } \pi \text{ of instances of}\\ \text{elements in a compound expression and}\\ \text{compound expression in array indexes}\end{array}\right)$$

as a potential function. It is obvious that every legal loop iteration decreases the value of this function F. ■

In the sequel, let us denote by $Sim(\pi)$ a simple program that is the result of application of the above simplification algorithm to a given mini-NIL(R, A)-program π.

5.2. Array elimination

Intuition behind array elimination in a simple program is also very simple: just emulate any static array by two sets of new variables with indexes 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.

First we have to introduce some special related notions for a proper formalization of the above intuition and a correct annotation of the algorithm below.

Declaration unfolding. Assume that $r_1, \ldots r_n$ and r are some unsigned integer constants, and a, $x_{0\ldots0}$, ... $x_{r_1\ldots r_n}$, $y_{0\ldots0}$, ... $y_{r_1\ldots r_n}$ are disjoint identifiers. Then let us say that the array declaration $(ARRAY\ a[r_1, \ldots r_n] : [0..r])$ unfolds into the set of declarations of value variables[8] $(VAR\ x_{0\ldots0} : [0..r])$, ... $(VAR\ x_{r_1\ldots r_n} : [0..r])$ and the set of declarations of indicating

[8]Value variable is a variable for representing the value of the corresponding array element. For example, variables $x0$, $x1$ and $x2$ from the above paragraph.

variables[9] $(VAR\, y_{0...0} : [0..1])$, ... $(VAR\, y_{r_1...r_n} : [0..1])$. To be short (when the declarations of the array and the separator symbol are implicit), let us say that the declaration of the array a unfolds to the set of declarations of the value variables $\overline{x's}$ and to the set of declarations of the indicating variables $\overline{y's}$.

Preamble unfolding. Let δ' and δ'' be two preambles such that δ'' results from δ' by unfolding some array declarations into the sets of declarations of fresh value and indicating variables. In this case, let us say that δ' unfolds onto δ'' by means of unfolding these array declarations to the corresponding sets of value and indicating variables.

State folding and unfolding. Let δ' be a preamble and δ'' be the result of unfolding some array declarations to corresponding declarations of sets of value and indicating variables. Let $\sigma' \in \Sigma(\delta')$ and $\sigma'' \in \Sigma(\delta'')$ be two states such that $\sigma'(t) = \sigma''(t)$ for every identifier t that is declared in both preambles δ' and δ'' as a variable or an array, but for every array declaration $(ARRAY\, a[r_1, \ldots r_n] : [0..r])$ in δ' that unfolds onto the set of declarations of value and index variables $(VAR\, x_{0...0} : [0..r])$, ... $(VAR\, x_{r_1...r_n} : [0..r])$ and $(VAR\, y_{0...0} : [0..1])$, ... $(VAR\, y_{r_1...r_n} : [0..1])$, the following holds for all integers $i_1 \in [0..r_1]$, ... $i_n \in [0..r_n]$:

- if $\sigma'(a[i_1, \ldots i_n])$ is indefinite, then $\sigma''(y_{i_1...i_n}) = 0$;
- if $\sigma'(a[i_1, \ldots i_n])$ is definite, then $\sigma''(y_{i_1...i_n}) = 1$ and $\sigma''(x_{i_1...i_n}) = \sigma'(a[i_1, \ldots i_n])$.

Then let us say that σ'' is unfolding of σ' (by means of unfolding the corresponding arrays onto the corresponding sets of value and indicating variables).

Unfolding function. For every preamble δ' and its every unfolding δ'' by means of unfolding some array declarations to the corresponding declarations of the sets of value and indicating variables, for every $\sigma' \in \Sigma(\delta')$ let $uf(\sigma',\ \delta',\ \delta'')$ be a state $\sigma'' \in \Sigma(\delta'')$ that is unfolding of σ' by means of unfolding the corresponding arrays onto the corresponding sets of value and indicating variables. Thus the function uf is defined; in the sequel, we apply this function uf to pairs of states, sets of states and sets of pairs of states in the component-wise manner.

Now we are ready to present an annotated pseudo-code of the algorithm that we will refer to as array elimination.

Precondition: [π is a simple mini-NIL(R, A) program with a preamble δ and a small step semantics S.]

Algorithm:

WHILE the preamble $P(\pi)$ of the program π has any array declaration DO

[9] Indicating variable is a variable for indicating whether the corresponding array element is defined. For example, variables $y0$, $y1$ and $y2$ from the above paragraph.

BEGIN
  LET ($ARRAY\ a[r_1, \ldots r_n]\ :\ [0..r]$) be
    an array declaration in the preamble $P(\pi)$;
  LET γ be a preamble that is the result of unfolding
    the declaration ($ARRAY\ a[r_1, \ldots r_n]\ :\ [0..r]$) in the preamble $P(\pi)$
    into the set of declarations of fresh value variables
      ($VAR\ x_{0\ldots0}\ :\ [0..r]$), ... ($VAR\ x_{r_1\ldots r_n}\ :\ [0..r]$)
    and the set of declarations of fresh indicating variables
      ($VAR\ y_{0\ldots0}\ :\ [0..1]$), ... ($VAR\ y_{r_1\ldots r_n}\ :\ [0..1]$);
  LET β be the program body $B(\pi)$;
  WHILE β has any update of any element of the array a OR
      any assignment of any element of the array a to a variable
DO
  BEGIN
    LET ε be an instance of
      an update of an element of the array a OR
      an assignment of an element of the array a to a variable;
    β := UNFOLD(ε, β); // See fig.5.2 for function UNFOLD.
  END;
  β := a program with the preamble γ and body β
END.

Postcondition:
[π is an array-free mini-NIL(R, A) program, and its small step semantics $SSS(\pi)$ is equal to $uf(S, \delta, \gamma)$.]

Proposition 3. *The array elimination algorithm is totally correct with respect to its precondition and postcondition.*

Proof (sketch). Let us use again the loop invariant and potential function techniques. Partial correctness can be proved as follows. We can adopt the following relaxation of the postcondition

$$\pi \text{ is a mini-NIL(R, A) program,}$$
$$\text{and its small step semantics } SSS(\pi) \text{ is equal to } uf(S, \delta, \gamma)$$

as the invariant of the external loop. It is straightforward that

- the precondition implies the invariant,
- the invariant and negation of the loop condition imply the post condition.

Invariance of this invariant over every legal iteration of the loop body follows from the following argument. It is sufficient to observe that a pair of

configurations $((l, \sigma),\ (l', \sigma'))$ is a firing of a 'simple' element update operator or an assignment of an element $(l :\ op)$ iff there exists a trace of UNFOLD$((l :\ op), \beta)$ that starts from $(l, uf(\sigma))$ and finishes in $(l', uf(\sigma'))$.

Termination can be proved by adopting the following mapping

$$F :\ \pi\ \mapsto \left(\begin{array}{l} \text{the total number of instances of array declarations,} \\ \text{element updates, and assignments of elements in } \pi \end{array} \right)$$

as a potential function. It is obvious that every legal loop iteration decreases the value of this function F. ■

In the sequel, let us denote by $Elm(\pi)$ the array-free program that is the result of application of the above elimination algorithm to a given simple program π in mini-NIL(R, A).

5.3. Range uniformation

The idea behind range uniformation is very trivial: if the range r a variable x is not equal to the $MaxInt$, then before any assignment to this variable $x := \tau$ test whether the value of τ is in the range $[0..r]$. Let us present an annotated pseudo-code of an algorithm that we will refer to as range uniformation.

Precondition: [π is a mini-NIL(R, A) array-free program, and S is its small-step semantics.]

Algorithm:

WHILE π has any variable declaration DO
BEGIN
  LET $(VAR\ x\ :\ [0..r])$ be a variable declaration within $P(\pi)$;
  π := REMOVE the declaration $(VAR\ x\ :\ [0..r])$ from π;
  WHILE π has any assignment to the variable x DO
  BEGIN
    LET $(l :\ x :=\ \tau\ goto\ L)$ be an instance of an assignment to x within π,
      and LET k be a fresh label;
    β := REPLACE the instance of $(l :\ x :=\ \tau\ goto\ L)$
      by the following couple of operators
      $l :\ if\ \tau \le r\ then\ \{k\}\ else\ \emptyset$
      $k :\ x :=\ \tau\ goto\ L$
  END
END.

Postcondition:

[π is a mini-NIL program, and its small step semantics $SSS(\pi)$ is equal to $S \cap (\Sigma(\pi))^2$.]

Proposition 4. *The range uniformation algorithm is totally correct with respect to its precondition and postcondition.*

FUNCTION UNFOLD
(ε: OPERATOR INSTANCE, β: PROGRAM BODY) : PROGRAM BODY
BEGIN
LET $l_{0\ldots0}$, ... $l_{i_1\ldots i_n}$, ... $l_{r_1\ldots r_n}$, $k_{0\ldots0}$, ... $k_{i_1\ldots i_n}$, ... $k_{r_1\ldots r_n}$, $m_{0\ldots0}$, ... $m_{i_1\ldots i_n}$, ... $m_{r_1\ldots r_n}$
be fresh disjoint labels;
CASE ε OF
an instance of some update ($l: \ a[t_1, \ldots t_n] := \ \tau \ goto \ L$):
replace ε in β by the following set of operators
$l: \ goto \ \{l_{0\ldots0}\}$
$l_{0\ldots0}: \ if \ t_1 = 0 \ \& \ldots \& \ t_n = 0 \ then \ \{k_{0\ldots0}\} \ else \ \{l_{0\ldots0\oplus1}\}$
$k_{0\ldots0}: \ x_{0\ldots0} \ := \ \tau \ goto \ \{m_{0\ldots0}\}$
$m_{0\ldots0}: \ y_{0\ldots0} \ := 1 \ goto \ L$
.....
$l_{i_1\ldots i_n}: \ if \ x_1 = i_1 \ \& \ldots \& \ x_n = i_n$
$then \ \{k_{i_1\ldots i_n}\} \ else \ \{l_{i_1\ldots i_n\oplus1}\}$
$k_{i_1\ldots i_n}: \ x_{i_1\ldots i_n} \ := \ \tau \ goto \ \{m_{i_1\ldots i_n}\}$
$m_{i_1\ldots i_n}: \ y_{i_1\ldots i_n} \ := 1 \ goto \ L$
.....
$l_{r_1\ldots r_n}: \ if \ x_1 = r_1 \ \& \ldots \& \ x_n = r_n \ then \ \{k_{r_1\ldots r_n}\} \ else \ \emptyset$
$k_{r_1\ldots r_n}: \ x_{r_1\ldots r_n} \ := \ \tau \ goto \ \{m_{r_1\ldots r_n}\}$
$m_{r_1\ldots r_n}: \ y_{r_1\ldots r_n} \ := 1 \ goto \ L$;
an instance of some assignment ($l: \ t := \ a[t_1, \ldots t_n] \ goto \ L$):
replace ε in β by the following set of operators
$l: \ goto \ \{l_{0\ldots0}\}$
$l_{0\ldots0}: \ if \ t_1 = 0 \ \& \ldots \& \ t_n = 0 \ then \ \{k_{0\ldots0}\} \ else \ \{l_{0\ldots0\oplus1}\}$
$k_{0\ldots0}: \ if \ y_{0\ldots0} = 1 \ then \ \{m_{0\ldots0}\} \ else \ \emptyset$
$m_{0\ldots0}: \ t := x_{0\ldots0} \ goto \ L$
.....
$l_{i_1\ldots i_n}: \ if \ x_1 = i_1 \ \& \ldots \& \ x_n = i_n$
$then \ \{k_{i_1\ldots i_n}\} \ else \ \{l_{i_1\ldots i_n\oplus1}\}$
$k_{i_1\ldots i_n}: \ if \ y_{i_1\ldots i_n} = 1 \ then \ \{m_{i_1\ldots i_n}\} \ else \ \emptyset$
$m_{i_1\ldots i_n}: \ t := x_{i_1\ldots i_n} \ goto \ L$
.....
$l_{r_1\ldots r_n}: \ if \ x_1 = r_1 \ \& \ldots \& \ x_n = r_n \ then \ \{k_{r_1\ldots r_n}\} \ else \ \emptyset$
$k_{r_1\ldots r_n}: \ if \ y_{r_1\ldots r_n} = 1 \ then \ \{m_{r_1\ldots r_n}\} \ else \ \emptyset$
$m_{r_1\ldots r_n}: \ t := x_{r_1\ldots r_n} \ goto \ L$
END

Figure. Definition of the function UNFOLD, where '$\oplus$' is the lexicographical '*next*' on n-tuples $[0..r_1] \times \ldots [0..r_n]$

In the sequel, let us denote by $Uni(\pi)$ the mini-NIL program that is the result of application of the above uniformation algorithm to a given array-free program π in mini-NIL(R, A).

Propositions 1, 2, 3, and 4 altogether imply the following theorem of transformational semantics correctness.

Theorem: $uf(SSS(\pi)) \cap (\Sigma(\rho))^2 = SSS(\rho)$, where π is a mini-NIL(R, A) program and ρ is $Uni(Elm(Simp(\pi)))$.

6. Concluding remarks: what's next?

In this paper, we gave a brief overview of the F@BOOL@ project and presented an intermediate layer of the project programming language mini-NIL with variable ranges and static arrays. This intermediate layer programming language is provided by operational and transformational semantics; correctness of the transformational semantics has been provided by manual proof-sketches. Below we present and motivate some future research directions.

1. Complete the manual proof-sketches and redo them with any mechanized proof-assistance.
2. Provide mini-NIL with variable ranges and static arrays by a correct transformation semantics for annotated programs.

We would like to quote Call For Papers of the 4rd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory[10] as a motivation of the first research direction:

> 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 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.

The second research direction is very natural for F@BOOL@, since verification of annotated programs is the primary goal of the project. The problem in this case is in a conflict between

[10] `http://www.seas.upenn.edu/s̃weirich/wmm/`

- second-order interpretation by partial functions of arrays in mini-NIL with variable ranges and static arrays,
- and transformational semantics of static arrays as collections of variables.

In terms of axiomatic semantics, this conflict means that the standard second-order axiom for array update $\{\psi_{upd(a,t,\tau)/a}\}(a[t] := \tau)\{\psi\}$ is not admissible any more, but $upd(a,t,\tau)$ has to be represented explicitly. In terms of the weakest precondition transformer [7], it means that for an array $WP(a[t] := \tau,\ \psi)$ is not $\psi_{upd(a,t,\tau)/a}$, while for an integer variable $WP(x := \tau,\ \psi)$ is $\psi_{\tau/x}$ as usual; $WP(a[t] := \tau,\ \psi)$ has to perform update explicitly and can be a very complicated formula.

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.et al. On the problem of computer language classification // Joint NCC&IIS Bulletin. Ser.: Computer Science. — 2008. — Iss. 28. — P. 1–29.

[3] Ball T., Cook B., Levin V., Rajamani S. K. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft // Lect. Notes Comput. Sci. — Berlin: Springer-Verlag, 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., and Majumdar R. The software model checker blast: Applications to software engineering // Int. J. on Software Tools for Technology Transfer. — 2007. — N 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. Vol. 19. — American Math. Society, Providence, R. I., 1967. — P. 19–32.

[9] Gries D. The Science of Programming. — New York: Springer Verlag, 1981. — 350 p.

[10] Hoare C. A. R. The verifying compiler: A grand challenge for computing research // Perspectives of Systems Informatics (PSI'2003). — Berlin: Springer-Verlag, 2003. — Vol. 2890. — P. 1–12.

[11] 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.