Вычислительные технологии, 2020, том 25, № 5, с. 91–106. © ФИЦ ИВТ, 2020 ISSN 1560-7534
Computational Technologies, 2020, vol. 25, no. 5, pp. 91–106. © FRC ICT, 2020 eISSN 2313-691X

DOI:10.25743/ICT.2020.25.5.008

Towards verification of scientific and engineering programs. The CPPS project

D. A. Kondratyev*, A. V. Promsky
A. P. Ershov Institute of Informatics Systems SB RAS, Novosibirsk, Russia
*Correspoding author: Kondratyev Dmitry A., e-mail: apple-66@mail.ru

Received June 26, 2020, revised August 5, 2020, accepted October 15, 2020.

Nowadays, when formal fundamentals of program verification are well studied, researchers concentrate their efforts on domain-specific methods for various classes of programs. However, it seems that the field of scientific and engineering applications still lacks attention. We would like to contribute to filling this gap through the development of the Cloud Parallel Programming System (CPPS). The goal of this project is to create a parallel programming system for Sisal programs. Deductive verification of Sisal programs is one the of important subgoals. Since the Cloud Sisal language is built on the basis of loop expressions, their axiomatic semantics is the basis of Hoare's logic for the Sisal language. The Cloud Sisal loop expressions, array construction expressions and array element replacement expressions enable efficiently executable computational or engineering mathematics programs. Thus, we believe that our axiomatic semantics for these types of expressions may present an interesting result.

Keywords: Cloud Sisal, deductive verification, Cloud-Sisal-kernel, C-lightVer, cloud parallel programming system, ACL2, loop invariant.

Citation: Kondratyev D.A., Promsky A.V. Towards verification of scientific and engineering programs. The CPPS project. Computational Technologies. 2020; 25(5): 91–106. DOI:10.25743/ICT.2020.25.5.008.

Introduction

The parallel programming environment is the main goal of the CPPS project [1]. The Cloud Sisal [2] serves as its input language.

From the very beginning of the CPPS project we consider deductive verification as the main checking approach [3]. The first version of verification module actually analyses an intermediate C representation, thus using the C-lightVer system [4] as a plugin. Apart from unnecessary complication such scheme depends on maintenance of C code generation [5] and maintenance of C-lightVer itself [6]. Now we develop a direct Cloud Sisal program verification module.

Such development consists of several steps. First, we define a representative subset of the Sisal together with its axiomatic semantics. This type of semantics is a set of axioms and proof rules for the language constructions [7]. During the consequent development steps we can add new constructions as well as corresponding axioms and rules. This process will culminate in axiomatic semantics for the whole Cloud Sisal. We call Cloud-Sisal-kernel the current subset of Cloud Sisal possessing axiomatic semantics.

To define axiomatic semantics of Cloud-Sisal-kernel we apply the weakest precondition approach [8] accompanied by specifications written in ACL2 (A Computational Logic for Applicative Common Lisp) [9]. Some wp-subformulas come out from translation of Cloud Sisal expressions into ACL2. An appropriate translator was introduced into CPPS. It is defined recursively while translation of constants and variables serves as a recursion base.

The implicit parallelism in Cloud Sisal is supported by means of array element replacement expressions, array construction expressions as well as the loop expressions [10]. The headers of the loop expressions introduce triplets for the loop variables. In effect, they are ranges of values assigned to variables when a loop executes. Moreover, the loop execution produces sequence of values of reducible expression on each iteration. The value of the whole loop expression results from application of special reductions to this sequence. Such loop expressions allow us to implement effectively the linear algebra operations for example.

As a formal basis for the weakest preconditions of loop expressions and array element replacement expressions we adopt another interesting approach — the symbolic method of verification of definite iterations [11]. It introduces a special replacement operation (function *rep*) symbolically representing application of a loop. The main advantage of *rep* is that it makes loop invariants unnecessary [12]. Traditionally these invariants are problematic feature of axiomatic semantics.

In this paper we represent axiomatic semantics of the expressions forms discussed above.

Let us note that natural operational semantics for Sisal was already developed quite ago [13]. Such semantics is appropriate as a formal language definitions, but serves poorly for verification. A good example of Hoare's logic for a functional language can be found in [14]. However, the F^* program verification considered there is based on using loop invariants. Let us consider the loop translation into recursive functions. The paper [15] describes a quite promising approach to loop modelling inside theorem prover. On other hand, their loops do not contain reducible expressions whereas Cloud Sisal does. Another interesting solution of the invariant problem is based on lemma-functions [16]. It has been implemented in verification system AstraVer [17]. This method uses a special form of specifications which may be considered as a disadvantage in comparison with our approach.

1. The loop and array expressions as a base of the Cloud Sisal

Unlike the majority of functional languages, the Cloud Sisal is based rather on loops than recursion [10]. Of course, recursion is still supported, but CPPS is based on effective implementation of array manipulations and the loop expressions. Their form is the main reason of using loop expressions, since it provides convenient ways of vectorization. Here we briefly overview them.

Triplet is a structure of the form [*lower boundary .. upper boundary .. step*]. It defines arithmetical progression of elements between given boundaries with fixed step.

Range is a structure based on Cartesian product of triplets:

$$var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross } \ldots var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } triplet_n,$$

where $var_1, var_2, \ldots, var_{n-1}, var_n$ are variables and $triplet_1, triplet_2, \ldots, triplet_{n-1}, triplet_n$ are triplets.

The array element replacement expression looks like:

$$source\_array[var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross } \ldots \\ var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } var_n \texttt{ in } triplet_n \ := \ expr]$$

where *source_array* is the original array name, var_1, ..., var_n are variables, $triplet_1$, ..., $triplet_n$ are triplets, *expr* is the replacing expression (possibly depending on $v_{1..n}$). The Cartesian product of triplets forms the range. Internally, this range is a set of array index tuples with lexical order. This array element replacement expression changes the original array so that each element indexed by range is replaced by the value of *expr*.

The array construction expression is defined as follows:

$$\texttt{array}[0 \,..\, hight_1,\ 0 \,..\, hight_2,\ \ldots,\ 0 \,..\, hight_{n-1},\ 0 \,..\, hight_n]\ \texttt{of}$$
$$[var_1\ \texttt{in}\ triplet_1\ \texttt{cross}\ var_2\ \texttt{in}\ triplet_2\ \texttt{cross}\ \ldots$$
$$var_{n-1}\ \texttt{in}\ triplet_{n-1}\ \texttt{cross}\ var_n\ \texttt{in}\ triplet_n\ :=\ expr_1;\ \texttt{else}\ :=\ expr_2],$$

where var_1, ..., var_n are variables, $triplet_1$, ..., $triplet_n$ are triplets, $expr_1$ is a replacing expression which may depend on these variables, whereas default expression $expr_2$ does not depend on them.

Conceptually such expression creates n-dimensional array with dimensions corresponding to $hight_1$, ..., $hight_n$. If the index of an array element belongs to the Cartesian product of triplets, then the element value is initialized by $expr_1$. Otherwise, its value is defined by $expr_2$.

Consider the loop expression controlled by a range:

$$\texttt{for}\ var_1\ \texttt{in}\ triplet_1\ \texttt{cross}\ var_2\ \texttt{in}\ triplet_2\ \texttt{cross}\ldots$$
$$var_{n-1}\ \texttt{in}\ triplet_{n-1}\ \texttt{cross}\ var_n\ \texttt{in}\ triplet_n\ \texttt{do}$$
$$\texttt{returns}\ reduction\ expr\ \texttt{end for}$$

where var_i and $triplet_j$ are variables and triplets correspondingly, *expr* is a reducible expression which can depend on these variables, *reduction* is a reduction. This loop iterates over Cartesian product of triplets. The value of the loop after certain iteration is the value of reduction applied to the value of *expr* on this iteration and to the loop value after previous iteration. Such loop form effectively implements associative and commutative operations over tuples and matrices.

The set of useful reductions includes:

- `array of` *expr* takes a reducible sequence of values of *expr* during loop iterations and forms the array;
- `value of` *expr* returns the last value of a reducible sequence of values of *expr*;
- `sum of` *expr* calculates the sum of a reducible sequence of values of *expr*;
- `product of` *expr* calculates the product;
- as the name suggests, `greatest of` *expr* returns the biggest value in a sequence;
- while `least of` *expr* returns the lowest one.

2. Symbolic method of verification of definite iterations

We adapt the method which was described in [11]. It allows us to eliminate the loop invariants when iterations over altered data structures are considered.

First, we define a general notion of structures of finite length. Let $memb(S)$ denote the multiset of elements of a structure S and $|memb(S)|$ is its power. For structure S we define the following operators:

- $empty(S) = true$ iff $|memb(S)| = 0$.
- $choo(S)$ returns an arbitrary element of $memb(S)$, if $\neg empty(S)$.

- $rest(S) = S'$, where S' is a structure of the type of S and $memb(S') = memb(S) \setminus \{choo(S)\}$, if $\neg empty(S)$.

Obviously sets, tuples, lists, strings, arrays, files and trees are typical examples of such data structures [6].

Finally, we define a loop statement of the form

$$\texttt{for } x \texttt{ in } S \texttt{ do } v := body(v, x) \texttt{ end}$$

where S is a data structure, x is a variable of type "element of S", v is a tuple of the loop variables excluding x, *body* represents some calculation which does not change x itself and also such calculation is finite for every $x \in memb(S)$. This requirement of termination results in some restrictions on that vague "some calculation". Namely, the loop body can only contain assignments, conditional statements (possibly nested) and loop break statements. We call such loop `for` a definite iteration.

The operational semantics of definite iterations is based on recursive definition. Let v_0 denote the initial values of variables from v. In order to express the work of a definite iteration we introduce the replacement operation $rep(v, S, body)$ such that

- if $empty(S)$, then $rep(v_0, S, body) = v_0$,
- if $\neg empty(S)$, then $rep(v_0, S, body) = body(rep(v_0, rest(S), body), choo(S))$.

It still may be unclear what is the meaning of such non-deterministic (note operation *choo*) expression of loops over data structures. The detailed discussion can be found in [11]. It is enough to say that definite iterations relieve us from the loop invariants which are traditionally considered as a great obstacle.

3. Modelling the Cloud Sisal constructs in ACL2

The input language of ACL2 [9] is an applicative and strictly functional dialect of Common Lisp. Since Cloud Sisal is also functional, we could translate every expression from Cloud Sisal to semantically equivalent composition of ACL2 instructions. Let us take a tour over some of them.

It is quite natural to model Cloud Sisal by lists, this moment does not need detailed explanations. However, let us note two operations over indexed sequences we will use often in the following sections. If i is an index and l is a list, then (*nth i l*) returns the value of the i-th element in l. If *expr* is an expression in ACL2, then (*update-nth i expr l*) is a new list coinciding with l everywhere except i-th element which is equal to *expr*. And, of course, multidimensional arrays are modelled by nested lists.

To define new types we use special constructs provided by ACL2 library *fty*. If e is a new type, we generate corresponding constructor (macro *make-e*). Let p be a field of the structure r of type e. Construction *make* simply enumerates field names and expressions. The field name is preceded by colon and succeeded by expression. Obviously, the fields of a structure created by *make* are initialized by values of corresponding expressions. If p is the singular field in a structure of type e, then (*make-e :p expr*) is a new structure r of type e and $r.p = expr$.

The macro $b*$ is appropriate to model instruction composition. In fact, it extends the ACL2 macro $let*$ which is convenient to define a nested *let*. Consider the common form of $let*$:

$$(let^* ((var_1\ term_1)\ ...\ (var_n\ term_n))\ body),$$

where var_i are variables (not necessarily distinct), *body* and $term_j$ are ACL2 expressions. This form is equivalent to the following:

$$(let\ ((var_1\ term_1))\ldots(let\ ((var_n\ term_n))\ body)\ ...).$$

Thus, association of variables val_i with the values of corresponding expressions $term_i$ is carried out consequently. The value of expressions becomes the value of the whole construction. Every pair (var_i $term_i$) is called *binding* and every var_i is called a *local* variable of $let*$.

Generally, construction $b*$ takes the form:

$$(b^*\ \langle list\text{-}of\text{-}bindings\rangle.\langle list\text{-}of\text{-}result\text{-}forms\rangle)$$

where $\langle list\text{-}of\text{-}result\text{-}forms\rangle$ is a list of ACL2 expressions. The value of b^* is defined by the value of the last expression in $\langle list\text{-}of\text{-}result\text{-}forms\rangle$. By analogy with let^* the *binding* operations are executed consequently. In general case construction *binding* is of the form

$$(\langle binder\text{-}form\rangle\ [\langle expression\rangle])$$

where $\langle binder\text{-}form\rangle$ is a construction b^*-*binder* and $\langle expression\rangle$ is an ACL2 expression. Since b^* extends let^*, then association of a variable with an expression value is a special case of *binding*. In this case the variable is local in b^*.

So, a variable is one of possible forms of construction $\langle binder\text{-}form\rangle$. Another possible form looks like (*when condition*), where *condition* is a Boolean expression in ACL2. Let the block $b*$ contain a binding ((*when condition*) *expression*). Whenever *condition* is true, all successive binding operations are rejected and the value of *expression* becomes value of $b*$.

We define ACL2 function *triplet* to model Cloud Sisal triplets. It takes lower boundary *low*, upper boundary *hight* and the step *step* as arguments and returns the list of values of arithmetical progression defined by triplet. If $low = hight$ then *triplet* returns one-element list (*low*). Being applied to an empty triplet, the function *triplet* returns the empty list *nil*.

We also define ACL2 functions *partition* and *cartesian_product* to model Sisal ranges and Cartesian products of triplets. Let us note that Sisal range generated by a singular triplet is modelled by application of function *partition* to this triplet. The idea is to split a set into single-element subsets. So *partition* takes a list as an argument and returns the list of one-element lists.

When several triplets generate a range we use function *cartesian_product*. It takes two lists as arguments. The returning value is the list of pairs where the first elements in pairs are from the first list and the second elements are from the second list. The order is as follows: we take the first element of the first list and couple it with all elements of the second list; then the second element of the first list is united with all elements of the second list; and so on. Thus, the function *cartesian_product* forms the list of tuples of triplet values in lexical order.

To model the array element replacement we generate function *update_elements_id* (*id* is unique identifier). As the original Cloud Sisal expression does, this function creates a new array so that elements indexed by a range are replaced by the value of expression depending on indices. The body of *update_elements_id* is produced when we translate the replacing expression itself into ACL2. Let us note that this expression may contain variables from the current scope or the context variables. In other words, it may depend on variables of enclosing expressions *let* or on values of the function arguments. It means we have to use

the context variables inside *update_elements_id* body. By the way, ACL2 does not support lambda-functions of closures. So we need to use additional function arguments to address the context variables.

The structure type *environment_id* is generated to model the context. The fields of this structure correspond to the context variables. To simplify construction of objects of type *environment_id* we generate definitions of functions *create_environment_id*. A more detailed discussion of the context definition in ACL2 can be found in [6].

The array construction expression is modelled in ACL2 in two steps: first, we create an array with default element values; then this array is being processed by array element replacement with indices from Cartesian product of triplets.

We defined the ACL2 function *create_array* to construct that initial array. The first argument of *create_array* is the length, the second one is initializing value (one and the same for all elements). Function returns the list of the given length with all elements initialized by the given value. Multiple applications of *create_array* can build up a multidimensional array.

After the initial (multidimensional) array is produced by *create_array* we generate the function *update_elements_identifier* to replace elements whose indices are given by Cartesian product of triplets. This function goes through elements of Cartesian product, evaluates the dependent replacement expression and use it to update elements of initial array.

Since we define axiomatic semantics of the loop expressions by means of symbolic replacement, we introduce ACL2 functions *rep_id*, where *id* is unique identifier. Let us note that in our case iteration goes over the list of tuples from Cartesian product of triplets. We use the standard Lisp functions *car* and *cdr* as functions *choo* and *rest* correspondingly. The body of *rep_id* is produced when we translate the loop expression into ACL2. The context problem for loop expressions and array constructors is resolved in the same manner as we did for array element replacements.

4. Translating reductions into ACL2

Let us consider the function *reduct2acl2* implementing translation of Cloud Sisal reductions into ACL2. The arguments are as follows: reduction name, reducible expression on the current iteration ($expr_1$) and the loop value just after previous iteration ($expr_2$). This function returns ACL2 string modelling reduction application to $expr_1$ and $expr_2$.

We also use auxiliary function *reduction_init* which takes reduction name and returns the default reduction value. Together these two functions generate body of the replacement operation *rep*.

Let us define *reduct2acl2* and *reduction_init* for all reduction forms:

- If *reduction* is `array of`, then

$$reduct2acl2(\texttt{array of},\ expr_1,\ expr_2) = (cons\ expr_1\ expr_2),$$
$$reduction\_init(\texttt{array of}) = nil.$$

  We model Cloud Sisal arrays via ACL2 lists. Since the default value of a reduction is empty list, then initial value of $expr_2$ is also empty list. The successive application of *cons* to reducible sequence adds new elements to the head of the list.
- If *reduction* is `value of`, then

$$reduct2acl2(\texttt{value of},\ expr_1,\ expr_2) = expr_1.$$

This reduction returns the first argument, so such modelling always results in the last value of reducible sequence.
The default value of such reduction depends on type of $expr_1$. It guarantees that for all cases (the default value case corresponds to recursion base) function *rep* returns values of the same type.

- If *reduction* is `sum of` *expr*, then

$$reduct2acl2(\texttt{sum of},\ expr_1,\ expr_2) = (+\ expr_1\ expr_2),$$
$$reduction\_init(\texttt{sum of}) = 0.$$

The default value of reduction is 0, so zero is also initial value of $expr_2$.

- If *reduction* is `product of` *expr*, then

$$reduct2acl2(\texttt{productof},\ expr_1,\ expr_2) = (*\ expr_1\ expr_2),$$
$$reduction\_init(\texttt{product of}) = 1.$$

The idea is analogous to reduction `sum of`.

- If *reduction* is `greatest of` *expr*, then

$$reduct2acl2(\texttt{greatest of},\ expr_1,\ expr_2) = (max\ expr_1\ expr_2).$$

The default value of this reduction depends on the lowest integer of specific bit representation. The compiler of CPPS defines such value.

- If *reduction* is `least of` *expr*, then

$$reduct2acl2(\texttt{least of},\ expr_1,\ expr_2) = (min\ expr_1\ expr_2).$$

5. Translating Cloud Sisal expressions into ACL2

We implement the recursive function *sisal2acl2* to translate Cloud Sisal expressions into ACL2.

We start with auxiliary function *context_variables*. It takes a Cloud Sisal expressions and returns the set of variable names that occur in expression. Its implementation uses some data provided by compiler of CPPS system. Another auxiliary function *context2vector* lexically orders this name set transforming it into the tuple. The function *context_length* evaluates the length of the name tuple. Finally, the function *context2string* produces the string representation of the name tuple in the form of variable names separated by commas, thus serving as a part of ACL2 code generator.

It should be noted that the syntax of a singular triplet is equal to the syntax of the range based on a singular triplet. So we also defined function *range2acl2* to translate Cloud Sisal ranges into ACL2.

Here we describe definition of translator *sisal2acl2* for several classes of Cloud-Sisal-kernel expressions.

5.1. Literals and variables

Here we consider several base forms.

If *base_expr* is either $false$ or $true$ then

$sisal2acl2(base\_expr) = boolean$

where *boolean* is *nil* or t correspondingly.

If *base_expr* is a decimal integer or a variable then

$sisal2acl2(base\_expr) = base\_expr.$

5.2. Conditional expression

If $if\_expr \equiv$ `if` *cond* `then` *p_branch* `else` *n_branch* `end if` then

$sisal2acl2(if\_expr) = (if\ sisal2acl2(cond)\ sisal2acl2(p\_branch)\ sisal2acl2(n\_branch)).$

5.3. Binary operations

If $binary\_expr \equiv arg1\ f\ arg2$ and $f \in \{+, -, *, /, <, >, <=, >=, =, \&, |, \hat{}\}$, then

$sisal2acl2(binary\_expr) = (f'\ sisal\_to\_acl2(arg1)\ sisal\_to\_acl2(arg2))$

where f' is a string from $\{+, -, *, floor, <, >, <=, >=, equal, and, or, xor\}$ depending on original f.

5.4. Array element access

If $index\_expr \equiv a[index_1,\ index_2, \ldots,\ index_n]$, where a is an n-dimensional array, $index_1, \ldots, index_n$ are Cloud Sisal expressions, then

$sisal2acl2(index\_expr)\ =$
$(nth\ sisal2acl2(index_n)\ (nth\ sisal2acl2(index_{n-1})$
$\ldots$
$(nth\ sisal2acl2(index_2)\ (nth\ sisal2acl2(index_1)\ a)\ \ldots\ )).$

5.5. Triplets

If *triplet_expr* is a triplet of the form [*low* .. *hight* .. *step*], then

$sisal2acl2(triplet\_expr) = (triplet\ sisal2acl2(low)\ sisal2acl2(hight)\ sisal2acl2(step)).$

5.6. Ranges

If $range\_expr = triplet_1$ `cross` $triplet_2$ `cross` $\ldots$ `cross` $triplet_n$, then definition is by induction.

If $n = 1$ then

$range2acl2(range\_expr)\ =\ (partition\ sisal2acl2(triplet_1))$

If $n > 1$ then

$range2acl2(range\_expr)\ =$
$(cartesian\_product\ sisal2acl2(triplet_1)$
$(cartesian\_product\ sisal2acl2(triplet_2)$
$\ldots$
$(cartesian\_product\ sisal2acl2(triplet_{n-1})\ sisal2acl2(triplet_n))\ \ldots\ )).$

5.7. An algorithm to generate definition of the structure containing the context variables

Let *context* stand for a set of the context variable names. The algorithm takes it as a parameter and performs two actions. First, it creates a structure of type *environment_id* with fields corresponding to variables. Second, it defines function *create_environment_id*, which creates an object of type *environment_id* with corresponding field values.

The generated code of *create_environment_id* looks like:

$(defun\ create\_environment\_id$
$\quad (context2vector(context)_1$
$\quad context2vector(context)_2$
$\quad \ldots$
$\quad context2vector(context)_{context\_length(context2vector(context))-1}$
$\quad context2vector(context)_{context\_length(context2vector(context))})$
$(make\text{-}environment\_id$
$\quad :context2vector(context)_1$
$\qquad context2vector(context)_1$
$\quad :context2vector(context)_2$
$\qquad context2vector(context)_2$
$\quad \ldots$
$\quad :context2vector(context)_{context\_length(context2vector(context))-1}$
$\qquad context2vector(context)_{context\_length(context2vector(context))-1}$
$\quad :context2vector(context)_{context\_length(context2vector(context))}$
$\qquad context2vector(context)_{context\_length(context2vector(context))})).$

By definition the function body is an application of macro *make-environment_id*. As you can see, the context variable names turn into field names.

5.8. Function rep generation algorithm

Parameters of this algorithm are as follows: 1) $var_1, \ldots, var_n$ are variables over a range; 2) *expr* is an expression which may depend on these variables; 3) *reduction* is the sort of applied reduction. We define the set *context* as

$$context = context\_variables(expr) \setminus \{var_1, \ldots, var_n\}.$$

This subtraction provides the set of context variables without variables over range.

The generated code of function *rep_id* looks like

$(defun\ rep\_id\ (range\_tuples\ environment)$
$\quad (b*\ ((when\ (endp\ range\_tuples))\ reduction\_init(reduction))$
$\qquad (tuple\ (car\ range\_tuples))$
$\qquad (var_1\ (car\ tuple))$
$\qquad (var_2\ (car\ (cdr\ tuple)))$
$\qquad \ldots$
$\qquad (var_{n-1}\ (car\ (cdr\ \ldots\ (cdr\ tuple)\ \ldots\ )))$
$\qquad (var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ tuple)\ \ldots\ ))))$
$\qquad (context2vector(context)_1$

$$
\begin{array}{l}
\quad environment.context2vector(context)_1) \\
\quad (context2vector(context)_2 \\
\quad\quad environment.context2vector(context)_2) \\
\quad \ldots \\
\quad (context2vector(context)_{context\_length(context2vector(context))-1} \\
\quad\quad environment.context2vector(context)_{context\_length(context2vector(context))-1}) \\
\quad (context2vector(context)_{context\_length(context2vector(context))} \\
\quad\quad environment.context2vector(context)_{context\_length(context2vector(context))})) \\
reduct2acl2(reduction, \\
\quad\quad sisal2acl2(expr), \\
\quad\quad (rep\_id\ (cdr\ range\_tuples))))).
\end{array}
$$

The binding of variable names $var_1, \ldots, var_n$ within block $b*$ allows us to use the outcome of translation of *expr* into Cloud Sisal without variable renaming. Moreover, the block $b*$ binds the context variables to the fields of an object containing the context itself. Here, such an object denoted as *environment* is created when the function *create_environment_id* is applied to the context variables.

If the list of index tuples of the array *indices_tuples* is empty, then function *rep_id* results in the default reduction value obtained by application of *reduction_init* to reduction name. Otherwise, *rep_id* iterates over the range tuple list using recursive calls, and for every range tuple it returns application of reduction modelling function to *expr* and to recursive call which corresponds to reduction value after previous iterations. The name of reduction modelling function results from application of *reduct2acl2* to reduction name.

5.9. Array element replacement definition

Parameters of this algorithm are the same as in subsection 5.8 except for reduction. The *context* is also defined by analogy. The modelling function is

$$
\begin{array}{l}
(defun\ update\_elements\_id\ (indices\_tuples\ environment\ source\_array) \\
\quad (b*\ ((when\ (endp\ indices\_tuples))\ source\_array) \\
\quad\quad\quad (indices\ (car\ indices\_tuples)) \\
\quad\quad\quad (var_1\ (car\ indices)) \\
\quad\quad\quad (var_2\ (car\ (cdr\ indices))) \\
\quad\quad\quad \ldots \\
\quad\quad\quad (var_{n-1}\ (car\ (cdr\ \ldots\ (cdr\ indices)\ \ldots\ ))) \\
\quad\quad\quad (var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ indices)\ \ldots\ ))))) \\
\quad\quad (context2vector(context)_1 \\
\quad\quad\quad environment.context2vector(context)_1) \\
\quad\quad (context2vector(context)_2 \\
\quad\quad\quad environment.context2vector(context)_2) \\
\quad\quad \ldots \\
\quad\quad (context2vector(context)_{context\_length(context2vector(context))-1} \\
\quad\quad\quad environment.context2vector(context)_{context\_length(context2vector(context))-1}) \\
\quad\quad (context2vector(context)_{context\_length(context2vector(context))} \\
\quad\quad\quad environment.context2vector(context)_{context\_length(context2vector(context))})) \\
\quad\quad (update\text{-}nth\ var_1 \\
\quad\quad (update\text{-}nth\ var_2
\end{array}
$$

```
...
(update-nth var_{n-1}
(update-nth var_n sisal2acl2(expr)
    (nth var_{n-1}
    ...
    (nth var_2
    (nth var_1
        (update_elements_id (cdr indices_tuples) source_array))) ...))))).
```

There is a certain similarity to definition of *rep_id*. Since we model multidimensional arrays by ACL2 lists, some overhead is inevitable. It takes form of composed functions of list element access (*nth*) and array update *update-nth*.

5.10. The loop expression

Let *loop_expr* stands for expression of the form

$$\texttt{for } var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross} \ldots$$
$$var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } var_n \texttt{ in } triplet_n \texttt{ do}$$
$$\texttt{returns } reduction\ expr \texttt{ end for}$$

where $var_1, \ldots, var_n$ are variables, $triplet_1, \ldots, triplet_n$ are triplets, *expr* is an expression which may depend on those variables and *reduction* is a reduction. In this case

```
sisal2acl2(loop_expr) =
(rep_id
    (reverse range2acl2(triplet_1 cross triplet_2 cross ... triplet_{n-1} cross triplet_n))
    (create_environment_id
        context2string(
            context2vector(
                context_variables(expr) \ {var_1, var_2, ... var_{n-1}, var_n})))),
```

where *id* is unique identifier. The function *rep_id* is governed by order in the list of Cartesian product of triplets. The function *reverse* is used to guarantee that the order of reductions is correct. The set subtraction operation removes the variables over range. The string representations of variables become arguments of *create_environment_id*. The algorithm which generates definition of *create_environment_id* has been already described in Section 5.7.

5.11. The array element replacement expression

Let *array_rep_expr* be an array element replacement expression of the form

$$source\_array[var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross } \ldots$$
$$var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } var_n \texttt{ in } triplet_n := expr],$$

defined in Section 1. Then

```
sisal2acl2(array_rep_expr) =
(update_elements_id
    (reverse range2acl2(triplet_1 cross triplet_2 cross ... triplet_{n-1} cross triplet_n))
    (create_environment_id
```

$$context2string(\\ context2vector(\\ context\_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))) \\ source\_array),$$

where *id* is unique identifier. The arguments of *update_elements_id* are similar to those in Section 5.10.

5.12. Array construction expression

Let *array_construct_expr* be an expression of the form

$$\texttt{array}[0\ ..\ hight_1,\ 0\ ..\ hight_2,\ \ldots,\ 0\ ..\ hight_{n-1},\ 0\ ..\ hight_n]\ \texttt{of} \\ [var_1\ \texttt{in}\ triplet_1\ \texttt{cross}\ var_2\ \texttt{in}\ triplet_2\ \texttt{cross}\ \ldots \\ var_{n-1}\ \texttt{in}\ triplet_{n-1}\ \texttt{cross}\ var_n\ \texttt{in}\ triplet_n\ :=\ expr_1;\ \texttt{else}\ :=\ expr_2],$$

where $var_1, \ldots, var_n$ are variables, $triplet_1, \ldots, triplet_n$ are triplets, $expr_1$ is a replacing expression possibly depending on these variables, whereas $expr_2$ is a default expression which does not depend on them. In this case

$$sisal\_to\_acl2(array\_construct\_expr) = \\ (update\_elements\_id \\ (reverse\ range2acl2(triplet_1\ cross\ triplet_2\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n)) \\ (create\_environment\_id \\ context2string(\\ context2vector(\\ context\_variables(expr_1) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))) \\ (create\_array\ sisal2acl2(hight_1) \\ (create\_array\ sisal2acl2(hight_2) \\ \ldots \\ (create\_array\ sisal2acl2(hight_{n-1}) \\ (create\_array\ sisal2acl2(hight_n)\ sisal2acl2(expr_2)))\ \ldots\ ))),$$

where *id* is a unique identifier. The arguments of *update_elements_id* resemble those of the function *rep_id* from Section 5.10.

6. Axiomatic semantics of Cloud-Sisal-kernel

We base our semantic research on Hoare's logic [7] with classical notions of Hoare triple $\{P\}\ S\ \{Q\}$ and of partial correctness of a program S w. r. t. its precondition P and postcondition Q [8].

The axiomatic semantics may differ depending on the derivation order. The Cloud Sisal itself and chosen symbolic method propose that backward strategy is more appropriate. Thus, the weakest precondition calculus is used. Let us note that for a given pair of program S and postcondition Q the weakest precondition $wp(S, Q)$ has two properties

- the triple $\{wp(S, Q)\}\ S\ \{Q\}$ is true;
- for any formula P the truth of the triple $\{P\}\ S\ \{Q\}$ implies $P \to wp(S, Q)$.

So, the verification is based on the fact that $\{P\}\ S\ \{Q\}$ is true iff $P \to wp(S,\ Q)$ is true.

To make wp-calculus of the loop expressions practical we need a small trick. Namely, we fix the term *result* in our specification language. It will correspond to the values of computable expressions.

With such fixed term axiomatic semantics for the loop expressions is straightforward. Let $R(y \leftarrow exp)$ denote substitution of exp for all free occurrences of variable y in R. Then

$$wp(loop\_expr\ Q) = Q(result \leftarrow sisal2acl2(loop\_expr)).$$

The weakest precondition for other expressions of Cloud-Sisal-kernel is defined analogously.

7. Study case

As an illustration consider the matrix elements summation program:

```
function sum_matrix_elements (a: array of (array of integer),
                     n, m: integer returns integer)
 for i in 0..n-1..1 cross j in 0..m-1..1 do
 returns sum of a[i, j] end for end function
```

It takes an integer matrix a with n rows and m columns.

We use the following ACL2 formula as a precondition:

```
(and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
```

Predicate *integerp* checks whether its argument is an integer. Predicate *integer-matrixp* does analogous checks for matrices. In ACL2 an integer matrix is implemented by a list of n lists, each of them is an integer list of length m. Formally, this predicate is defined by a set of domain specific lemmas.

Postcondition is quite short:

```
(= result (sum-matrix n m a))
```

Here, *sum-matrix* is another function defined in ACL2 by lemmas.

When we translate loop expression under discussion into ACL2, it is modelled by application of function *rep_1*:

```
(rep_1 (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
       (create_environment_1 a))
```

and the definitions of *rep_1* and *create_environment_1* are also generated automatically:

```
(defun create_environment_1(a)
    (make-environment_1 :a a))

(defun rep_1(range_tuples environment)
  (b* ((when (endp range_tuples) 0)
       (tuple (car range_tuples))
       (i (car tuple))
       (j (car (cdr tuple)))
       (a environment.a))
    (+ (nth i (nth j a)) (rep_1 (cdr range_tuples) environment))))
```

Let us note that if we omit the array range variables then context variable set of expression $a[i, j]$ consists of a itself. Also function rep_1 corresponds to actual evaluation of the loop. Thus it defines an operational semantics.

To derive the weakest precondition for *sum-matrix-elements* and its postcondition every occurrence of the term *result* in the postcondition is replaced by translation outcome. This process leads to the following:

```
(= (rep_1 (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
          (create_environment_1 a))
   (sum-matrix n m a))
```

Hence, the function *sum-matrix-elements* is partially correct w.r.t. its annotations when the following formula is true:

```
(implies
 (and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
 (= (rep_1 (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
           (create_environment_1 a))
    (sum-matrix n m a)))
```

The antecedent of this implication is precondition of *sum-matrix-elements*. The weakest precondition for *sum-matrix-elements* and its postcondition form the consequent of this implication. ACL2 successfully proved it by induction on n and m. During the proof we also use lemmas about functions *integer-matrixp* and *sum-matrix*. Thus, function *sum-matrix-elements* corresponds to its specifications.

Conclusion

Here we discussed the current situation in CPPS project. We have considered axiomatic semantics of Cloud-Sisal-kernel. Our semantics is based on translation from Cloud Sisal into ACL2. As a result we construct axioms and rules for several forms of expressions of Cloud Sisal. Such expressions enable efficiently executable computational or engineering mathematics programs [10]. Since they are the core of Cloud Sisal, the subset Cloud-Sisal-kernel is quite representative.

An interesting side effect also occurs. Since functions *rep* and *update_elements_id* in fact define evaluation of corresponding expressions, we simultaneously devised operational semantics.

As for the future work, we may mention one limitation of Cloud-Sisal-kernel. At the moment we do not support the *while* section of loop expressions. This construction allows us to terminate a loop when certain condition is satisfied. Obviously, axioms and rules for general loop expressions will be an important step towards full coverage of Cloud Sisal.

Acknowledgements. This work was carried out with a grant from the Russian Science Foundation (project 18-11-00118).

References

[1] **Kasyanov V., Kasyanova E.** Methods and system for cloud parallel programming. Proceedings of the 21st International Conference on Enterprise Information Systems, Heraklion, 2019. Setubal: SciTePress; 2019: 623–629.

[2] **Kasyanov V., Kasyanova E.** A system of functional programming for supporting of cloud supercomputing. WSEAS Transactions on Information Science and Applications. 2018; 15(9):81–90.

[3] **Hähnle R., Huisman M.** Deductive software verification: From pen-and-paper proofs to industrial tools. Lecture Notes in Computer Science. 2019; (10 000):345–373.

[4] **Kondratyev D., Promsky A.** Proof strategy for automated Sisal program verification. Lecture Notes in Computer Science. 2019; (11 771):113–120.

[5] **Kondratyev D., Promsky A.** Correctness of proof strategy for the Sisal program verification. Proceedings of 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON), Novosibirsk, 2019. IEEE; 2019: 641–646.

[6] **Kondratyev D., Maryasov I., Nepomniaschy V.** The automation of C program verification by the symbolic method of loop invariant elimination. Automatic Control and Computer Sciences. 2019; 53(7):653–662.

[7] **Hoare C.A.R.** An axiomatic basis for computer programming. Communications of the ACM. 1969; 12(10):576–80.

[8] **Apt K.R., Olderog E.-R.** Fifty years of Hoare's logic. Formal Aspects of Computing. 2019; 31(6):751–807.

[9] **Moore J.S.** Milestones from the pure Lisp theorem prover to ACL2. Formal Aspects of Computing. 2019; 31(6):699–732.

[10] **Kasyanov V.** Sisal 3.2: functional language for scientific parallel programming. Enterprise Information Systems. 2013; 7(2):227–236.

[11] **Nepomniaschy V.** Symbolic method of verification of definite iterations over altered data structures. Programming and Computer Software. 2005; (31):1–9.

[12] **Kondratyev D.A.** The extension of the C-light project using symbolic verification method of definite iterations. Computational Technologies. Special Issue: "XVII All-Russian Conf. of Young Scientists on Mathematical Modeling and Information Technology". 2017; (22):44–59. (In Russ.).

[13] **Attali I., Caromel D., Wendelborn A.** A formal semantics and an interactive environment for Sisal. International Series in Software Engineering. 1996; (2):229–256.

[14] **Swamy N., Hriţcu C., Keller C., Rastogi A., Delignat-Lavaud A., Forest S., Bhargavan K., Fournet C., Strub P.-Y., Kohlweiss M., Zinzindohoue J.-K., Zanella-Béguelin S.** Dependent types and multi-monadic effects in F^*. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, USA, 2016. New York: ACM; 2016: 256–270. Available at: `https://www.fstar-lang.org/papers/mumon/paper.pdf`

[15] **Myreen M.O., Gordon M.J.C.** Transforming programs into recursive functions. Electronic Notes in Theoretical Computer Science. 2009; (240):185–200.

[16] **Volkov G., Mandrykin M., Efremov D.** Lemma functions for Frama-C: C programs as proofs. Proceedings of 2018 Ivannikov Ispras Open Conference (ISPRAS), Moscow, 2018. IEEE; 2018: 31–38.

[17] **Efremov D., Mandrykin M., Khoroshilov A.** Deductive verification of unmodified Linux Kernel library functions. Lecture Notes in Computer Science. 2018; (11 245):216–234.

Вычислительные технологии, 2020, том 25, № 5, с. 91–106. © ФИЦ ИВТ, 2020 ISSN 1560-7534
Computational Technologies, 2020, vol. 25, no. 5, pp. 91–106. © FRC ICT, 2020 eISSN 2313-691X

INFORMATION TECHNOLOGIES

DOI:10.25743/ICT.2020.25.5.008

На пути к верификации научных и инженерных программ. Проект CPPS

Д. А. Кондратьев*, А. В. Промский

Институт систем информатики им. А. П. Ершова СО РАН, Новосибирск, Россия

* Контактный автор: Кондратьев Дмитрий Александрович, e-mail: apple-66@mail.ru

Поступила 26 июня 2020 г., доработана 5 августа 2020 г., принята в печать 15 октября 2020 г.

Аннотация

В настоящее время, когда теоретические основы верификации программ хорошо изучены, исследователи концентрируют свои усилия на предметно-ориентированных методах для различных классов программ. Инструменты, которые они выбирают, варьируются от проверки моделей для сетевых протоколов до исчислений указателей для фрагментов ядра операционной системы. Однако, похоже, что области научных и инженерных программ все еще уделяется недостаточно внимания. Мы хотели бы внести свой вклад в заполнение этого пробела с помощью разработки системы CPPS. Целью этого проекта является создание системы параллельного программирования для Sisal-программ. Дедуктивная верификация Sisal-программ является одной из важных подцелей. Так как язык Cloud-Sisal построен на основе циклических выражений, их аксиоматическая семантика является базой логики Хоара для языка Sisal. Циклические выражения языка Cloud-Sisal, выражения конструирования массивов и выражения замещения элементов массивов позволяют реализовать эффективно исполняемые программы вычислительной или инженерной математики. Таким образом, мы полагаем, что наша аксиоматическая семантика для этих типов выражений может представлять интересный результат. Природа таких программ позволяет достичь не только эффективного исполнения, но и упростить верификацию. Действительно, программы вычислительной математики часто основаны на итерациях над структурами данных. Символический метод верификации финитных итераций является в этой ситуации очень полезным, так как он элиминирует те проблемные инварианты цикла, которые всегда мешают формальной верификации. Все предыдущие исследования этого метода были теоретическими, CPPS представляет собой первую попытку использования его на практике.

Ключевые слова: Cloud-Sisal, дедуктивная верификация, Cloud-Sisal-kernel, C-lightVer, облачная система параллельного программирования, ACL2, инвариант цикла.

Цитирование: Kondratyev D.A., Promsky A.V. Towards verification of scientific and engineering programs. The CPPS project. Computational Technologies. 2020; 25(5):91–106. DOI:10.25743/ICT.2020.25.5.008.

Благодарности. Исследование выполнено при поддержке Российского научного фонда (грант № 18-11-00118).