Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 229 (2009) 67–84
www.elsevier.com/locate/entcs

Formalisation of C Language Interfaces

Gustavo A. Ospina[1]

Faculty of Computer Science
University of Namur
Belgium

Baudouin Le Charlier[2]

Department of Computer Science and Engineering
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Abstract

In practical computing, implementations of programming languages provide an interface that allow programs written in a language to call code written in another programming language, most often C. Usually, those language interfaces are left out of the formal definition of the language, and reasoning about multi-language programs is very difficult due to the lack of precise specifications for the language interfaces. In this paper, we present an application of a framework for the interoperability of programming languages, in which we specified in a systematic way the C interface of a real, rule-based programming language. Our framework is based on simple combinations of the small-step operational semantics of programming languages. We give the main elements of a small-step semantics for the C programming language that can be used to specify the same kind of interfaces for other programming languages implemented in C.

Keywords: interoperability, programming languages, operational semantics, foreign language interfaces.

1 Introduction

A very important aspect of the actual global society is the free access to information from anywhere in the world. It makes easier the exchanges and the agreements between individuals and enterprises, which can decide to work together in order to carry out their respective goals. Computing systems must adapt to those environments, and a non negligible task for software developers is to adapt the existing systems to make them work together, that is, to make them *interoperate*.

Interoperability is a practical issue which arises in all the contexts of computing, from complex information systems to single programs. Usually, the entities we want

[1] Email: gos@info.fundp.ac.be
[2] Email: Baudouin.Lecharlier@uclouvain.be

doi:10.1016/j.entcs.2009.07.074

to make interoperate do not have been designed to work together. This is mostly true for programming languages, which are closed representations of an algorithmic world. To make programs written in different languages to interoperate, some "glue code" must be added in order to make the corresponding object modules compatible. A practical way to write that code is using a *foreign language interface* (FLI) which defines a set of directives or built-in procedures for calling code implemented in another (foreign) programming language. In many cases, the FLI relates a high level language to the low level language used for its implementation. Thus, programmers have access to internal details of the language implementation and built language extensions directly in the low level language. C is the best example of a low level language used in the implementation of many well known high level languages. Since C is close to the operating system and most of existing libraries are written in C, almost all the implementations of practical programming languages provide a "C interface".

In this paper, we present the formal definition of a C interface for a real, rule-based programming language, Russel [4]. This is an application of the conceptual framework for programming languages interoperability defined in [9][10], which is based on the combination of the small-step operational semantics of interoperating languages. We present the difficulties to reason about mixed Russel-C code, because of potential side effects of the C code on the Russel implementation. Our framework overcomes those difficulties by a precise description of how inter-language calls affect the execution in both sides of a Russel-C program, and we can still use the reasoning mechanisms given by the original semantics of Russel and C.

The essentials of our programming language interoperability framework are shown in next section. Then, we present the main elements for a complete operational semantics of C, and especially its memory model. Afterwards, we show the application of our framework to a description of a C interface for a real programming language, and we give some directions for our further work before concluding.

2 Our framework for programming languages interoperability

We developed a conceptual framework [9] in which the notion of interoperability is defined for programming languages. Roughly speaking, two programming languages interoperate if there is a mechanism that allows programs built in one language to interoperate with programs built in the other language. To make precise the notion of program interoperability at the language level, we classified different mechanisms for program interoperability in a taxonomy [10]. This paper will focus on foreign language interfaces (FLI) where C is the "foreign language".

Interoperating programs can be executed by different processes in the same machine or in different machines, or by a unique process. The later case is proper to C interfaces, where both C code and language code are compiled into the same object language. In [9][10] we mentioned three kinds of information exchange that can be supported by a FLI for each execution context.

- *Single data communication:* the FLI consists on single primitives for sending and receiving data to and from external programs. Interoperating programs are executed in separated processes.
- *Partial memory copying:* the FLI has just a primitive for calling foreign procedures. When a procedure is called, the memory zone occupied by actual parameters is "reflected" (copied) into the foreign memory. When the foreign procedure returns, that memory zone is "dereflected" (updated) with the changes made by the procedure. This mechanism is mostly used in FLI where languages are related by the implementation.
- *Foreign reference encoding:* the FLI has a primitive for calling foreign procedures, but also primitives for handling "foreign references" to data in the foreign memory. Those primitives include creation, deletion and fetch of the numerical value of a foreign reference. Calls to foreign procedures just send foreign references and numerical data as parameters. Some C interfaces like the Java Native Interface [6] mix foreign reference encoding (used by C code calling Java methods) and partial memory copying (used by Java methods that are actually implemented in C). This kind of mechanism can also be used in the context of separated processes, typically in Remote Procedure Call (RPC) mechanisms.

Our framework includes a systematic methodology to describe the execution of programs written in two different programming languages. This methodology has five stages:

(i) *Specification of small-step operational semantics.* The first stage is the specification of small-step operational semantics for both languages. We need a small-step semantics instead of a big-step semantics because transitions in small-step semantics are bound on time and we can suppose there is no way to have interoperability in the middle of a transition. It makes easier to combine the resulting operational semantics, as we will see in the last stage.

At the end of this stage we have precise definitions of program *configurations* for both languages. We describe a configuration as the composition of a *control* component and a *data* component, denoted by $\langle \sigma, \delta \rangle$. To distinguish program configurations on language A from configurations on language B, we use subscripts, as in $\langle \sigma_A, \delta_A \rangle$.

(ii) *Extension of languages syntaxes.* The second stage is the extension of syntaxes for both languages with interoperability constructs. Those constructs depend on the chosen exchange mechanism and define the concrete support to interoperability provided by the languages.

(iii) *Definition of an exchange data structure.* Afterwards, we define an "exchange data structure" that will be added to program configurations for each language. That structure mix data and control elements that are proper to the concrete exchange mechanism and is intended to facilitate the effective exchange (of data and procedure calls) between interoperating programs. We note an exchange data structure as γ, and the resulting extended configuration as $\langle \sigma + \gamma, \delta \rangle$. The $+$ operator is just intended to mean a kind of union of an element into

one of the control or data components of a configuration. Operations of data conversion between data representations in both languages should be specified at the end of this stage.

(iv) *Transition rules for interoperability constructs.* At this point, we can specify the transition rules for each interoperability construct. Those rules show how the exchanges are effectively carried out and how the execution of an interoperability construct affects the program configuration. A transition rule on extended configurations will have this form:

$$\frac{Conds_{A_{\text{Ext}}}}{\langle \sigma_A + \gamma, \delta_A \rangle \longrightarrow \langle \sigma'_A + \gamma', \delta'_A \rangle}$$

(v) *Combination of operational semantics.* Now we can build a "combined" operational semantics which describes the execution of a program with two components, written in languages A and B respectively.

Next subsections will give an abstract description of the techniques we developed to obtain the combined operational semantics depending on whether interoperating languages are related by the implementation.

2.1 Interoperability of languages with independent implementations

At this time, we have built two operational semantics for languages A and B with no relation between each other. In such case, we say that A and B are at the same level of abstraction. Concretely, this means that implementations of A and B are independent, and without any interoperability, there is no way to affect the execution of a program written in A by a program written in B.

Assuming that implementations of languages A and B and independent, we note a combined configuration as $\langle \sigma_A + \sigma_B + \gamma, \delta_A + \delta_B \rangle$. Transition rules of the combined operational semantics result from the application of following "meta-rules":

Definition 2.1 *Meta-rule for preserving the semantics of A.* Each transition rule of the operational semantics for language A with the form:

$$\frac{Conds_A}{\langle \sigma_A, \delta_A \rangle \longrightarrow \langle \sigma'_A, \delta'_A \rangle}$$

is translated into a rule of the form:

$$\frac{Conds_A}{\langle \sigma_A + \sigma_B + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma'_A + \sigma_B + \gamma, \delta'_A + \delta_B \rangle}$$

This meta-rule ensures that A components will be executed according to A semantics.

Definition 2.2 *Meta-rule for preserving the semantics of B.* Each transition rule

of the operational semantics for language B with the form:

$$\frac{Conds_B}{\langle \sigma_B, \delta_B \rangle \longrightarrow \langle \sigma'_B, \delta'_B \rangle}$$

is translated into a rule of the form:

$$\frac{Conds_B}{\langle \sigma_A + \sigma_B + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma_A + \sigma'_B + \gamma, \delta_A + \delta'_B \rangle}$$

This meta-rule ensures that B components will be executed according to B semantics.

Definition 2.3 *Meta-rule for preserving the semantics of "extended" A.* Each transition rule of the operational semantics for language A extended with interoperability constructs which has the form:

$$\frac{Conds_{A_{\text{Ext}}}}{\langle \sigma_A + \gamma, \delta_A \rangle \longrightarrow \langle \sigma'_A + \gamma', \delta'_A \rangle}$$

is translated into a rule of the form:

$$\frac{Conds_{A_{\text{Ext}}}}{\langle \sigma_A + \sigma_B + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma'_A + \sigma_B + \gamma', \delta'_A + \delta_B \rangle}$$

Generated rules describe how the execution of an interoperability construct on an A component affects the combined configuration.

Definition 2.4 *Meta-rule for preserving the semantics of "extended" B.* Each transition rule of the operational semantics for language B extended with interoperability constructs which has the form:

$$\frac{Conds_{B_{\text{Ext}}}}{\langle \sigma_B + \gamma, \delta_B \rangle \longrightarrow \langle \sigma'_B + \gamma', \delta'_B \rangle}$$

is translated into a rule of the form:

$$\frac{Conds_{B_{\text{Ext}}}}{\langle \sigma_A + \sigma_B + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma_A + \sigma'_B + \gamma', \delta_A + \delta'_B \rangle}$$

Generated rules describe how the execution of an interoperability construct on a B component affects the combined configuration.

2.2 Interoperability of a language and its implementation language

In many practical cases, a FLI is designed between a high level programming language A and the lower level language B used in the implementation of A. This is the case of C interfaces provided by most well-known implementations of programming languages. Semantics of interoperating languages are not in the same abstraction

level since one language is implemented in the other language. In the context of the interoperability of two programs written in A and B, this implies:

- There is a concrete representation of the A program configuration and the A execution model in the B program configuration.
- Applying a transition rule of A semantics on the A program configuration affects the data component in the B program configuration.
- Applying a transition rule of B semantics on the B program configuration can accidentally corrupt the A program configuration or even crash the A execution model, if that transition rule attempts to modify the memory zones on the B configuration used by the A implementation.

Starting from an abstract operational semantics for A, we need to "refine" that semantics to reflect minimal details of its implementation in B. We refine the A configuration by adding an exchange data structure and the data component of B configuration. This is noted by $\langle \sigma_A + \gamma, \delta_A + \delta_B \rangle$. Then, we specify a coherence condition for abstract A configurations with the predicate $\texttt{CoherentConf}(\langle \sigma_A, \delta_A \rangle, \delta_B)$, which is true iff the abstract A configuration and the A execution model are correctly implemented in the B data store. Instead of meta-rules 2.1 and 2.3, we use the following meta-rules for refining the A semantics:

Definition 2.5 *Meta-rule for refining the semantics of A.* Each transition rule of the operational semantics for language A with the form:

$$\frac{Conds_A}{\langle \sigma_A, \delta_A \rangle \longrightarrow \langle \sigma'_A, \delta'_A \rangle}$$

is *refined* into a rule of the form:

$$\frac{Conds_{A_{\text{Ref}}} \wedge \texttt{CoherentConf}(\langle \sigma_A, \delta_A \rangle, \delta_B)}{\langle \sigma_A + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma'_A + \gamma', \delta'_A + \delta'_B \rangle}$$

Refined rules must ensure the new A configuration is coherent, this is, $\texttt{CoherentConf}(\langle \sigma'_A, \delta'_A \rangle, \delta'_B)$. In order to achieve this, some conditions might be added on the added components (γ', δ'_B).

Definition 2.6 *Meta-rule for preserving the semantics of "refined" A.* Each transition rule of the refined semantics of A with the form:

$$\frac{Conds_{A_{\text{Ref}}}}{\langle \sigma_A + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma'_A + \gamma', \delta'_A + \delta'_B \rangle}$$

is translated into a rule of the form:

$$\frac{Conds_{A_{\text{Ref}}}}{\langle \sigma_A + \sigma_B + \gamma, \delta_A + \delta_B \rangle \longrightarrow \langle \sigma'_A + \sigma_B + \gamma', \delta'_A + \delta'_B \rangle}$$

$$
\begin{array}{rcl}
type & ::= & \texttt{byte} \mid \texttt{int} \mid \texttt{struct st} \mid type * \\
expr & ::= & \mathsf{n} \mid \mathsf{str} \mid \mathsf{arr} \mid lexp \mid lexp.\mathsf{farr} \mid \& \; lexp \mid uop \; expr \mid expr \; bop \; expr \\
lexp & ::= & \mathsf{v} \mid expr[expr] \mid lexp.\mathsf{fv} \\
uop & ::= & - \mid ! \\
bop & ::= & + \mid - \mid * \mid / \mid \% \mid \& \mid '|' \mid \&\& \mid '||' \mid == \mid < \mid > \\
stmt & ::= & lexp \; = \; expr \\
 & \mid & \mathsf{fn}(expr*) \mid lexp \; = \; \mathsf{fn}(expr*) \\
 & \mid & \texttt{return} \mid \texttt{return} \; expr \\
 & \mid & lexp \; = \; \texttt{malloc}(expr, type) \mid \texttt{free}(expr) \\
 & \mid & \texttt{read}(lexp) \mid \texttt{print}(expr) \\
 & \mid & \{stmt; *\} \\
 & \mid & \texttt{if}(expr) \; stmt \mid \texttt{if}(expr) \; stmt \; \texttt{else} \; stmt \\
 & \mid & \texttt{while}(expr) \; stmt
\end{array}
$$

Fig. 1. C-like EBNF syntax for expressions and statements

Meta-rules given in definitions 2.2 and 2.4 are still applicable for language B, even though the predicate $\texttt{CoherentConf}(\langle\sigma_A, \delta_A\rangle, \delta_B)$ should be verified for each generated rule.

3 Towards a small-step operational semantics for a C-like language

Since a complete operational semantics for the C language [5] is too large to be presented here, in this section we just show the main elements of that semantics which are used in the definition of C interfaces for programming languages. French speaking readers can find a complete description in [9].

3.1 Differences with "real" C

Our C-like language has some differences with respect to the full definition of C. First, our language has only two numerical types: `byte` and `int`, and it has neither unions nor function pointers. Second, some statements were omitted from our definition: `switch`, `for`, `do`, `break`, and `continue`. Instead, we add statements for I/O and memory management: `read`, `write`, `malloc`, and `free`. Fig. 1 shows a fragment of the C-like syntax we defined for types, expressions and statements. In that syntax, n represents a numerical constant, str a string constant, arr an array name, st a struct name, fn a function name, v a variable name, fv a struct variable field name, and farr a struct array field name.

3.2 Memory model and data representation

We kept the memory model for our C-like language simple and close to the C standard specification [1]. This model handles both abstract values and their "low level" representation as sequences of bytes. The starting point is the definition of three sets: Byte is an abstract set of distinct values which can be represented in one byte (e.g. the interval `0..255`). $\mathsf{Int_C}$ is the set of all the integers in the C type `int`, i.e. the interval `MININT..MAXINT`. The set of *memory locations*, $\mathsf{Loc_C}$, is defined as

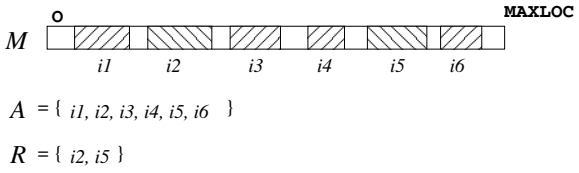

Fig. 2. View of a C store

the interval 0..MAXLOC. The constants MININT, MAXINT and MAXLOC are defined by the concrete implementation. The location 0 is also known as NULL.

Byte values fit in only one memory location. Values representing memory locations have a low level representation as a sequence of SIZELOC bytes. This is the length of the interval of locations needed to store a value. In a similar way, any integer value can be represented in a sequence of SIZEINT bytes. An array of n elements of type t are filled in an interval of $n * \texttt{sizeof}(t)$ locations. C structs are filled in an interval of locations which is big enough to contain all their fields, such that each field satisfies its alignment constraint, as it is defined in the C language standard [1].

Formally, let $\mathsf{ILoc_C}$ be the set of all the intervals of locations in $\mathsf{Loc_C}$. We define a *memory* as a function that binds memory locations to bytes. Let $\mathsf{Mem_C} \triangleq \mathsf{Loc_C} \rightarrow \mathsf{Byte}$ be the set of memories. We precise the memory zones that can be accessed by a C program under the notion of *C store*. We define a C store as a tuple $\langle M, A, R\rangle$ where M is a memory and A, R two sets of locations intervals. A represents the set of memory locations currently allocated by the C program for storing data. R is a subset of A containing the memory locations marked as "read-only". We denote a C store by S and its components by $S.A, S.M$ and $S.R$. Let $\mathsf{Store_C}$ be the set of C stores. A graphical view of a C store is shown in fig. 2.

Each C store contains a low level abstract representation of the whole computer memory. It is well known that the operating system allocates intervals of memory locations for each running process, where both data and object code is stored. In the store associated to a C program, we specified the zones of memory that were allocated to store static and dynamic data and our operational semantics constrains the C program to access only those zones. The rest of the memory can be used by the concrete C implementation, the operating system or other programs. To be able to reason about the changes made by a C program on its store without needing to be aware of the contents of the whole computer memory, we defined an equivalence relation on C stores: $S \cong_C S'$ iff $S.A = S'.A, S.R = S'.R$ and $S.M(l) = S'.M(l)$ for each location $l \in il \in S.A$. In a similar way, we defined an order relation over C stores: $S \succ_C S'$ iff $S.A \supset S'.A, S.R = S'.R$ and $S.M(l) = S'.M(l)$ for each location $l \in il \in S'.A$. Those relations will be useful to specify low level operations in C programs:

$$
\begin{array}{lll}
\texttt{fetchStore} & : \mathsf{Store_C} \times \mathbb{N} \times \mathbb{N} \times \mathsf{Loc_C} & \rightarrow (\mathsf{Store_C} \times \mathsf{Byte}^*) \uplus \{error\} \\
\texttt{updateStore} & : \mathsf{Store_C} \times \mathbb{N} \times \mathsf{Loc_C} \times \mathsf{Byte}^* & \rightarrow \mathsf{Store_C} \uplus \{error\} \\
\texttt{allocInStore} & : \mathsf{Store_C} \times \mathbb{N} \times \mathbb{N} & \rightarrow \mathsf{Store_C} \times \mathsf{Loc} \\
\texttt{freeInStore} & : \mathsf{Store_C} \times \mathsf{Loc_C} & \rightarrow \mathsf{Store_C} \uplus \{error\}
\end{array}
$$

$\texttt{fetchStore}(S, n, m, l)$ gets n bytes from the location l (which must be a multiple of m) in store S. The fetched locations must be included in an interval of $S.A$. $\texttt{updateStore}(S, m, l, bs)$ assigns the sequence of bytes bs to store S starting from location l. The number of affected locations is equal to the length of bs, the affected

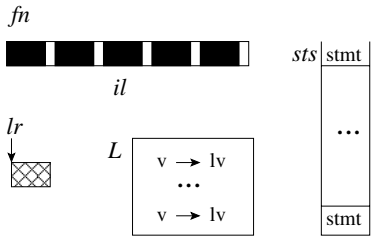

Fig. 3. View of a C function environment

locations must be included in $S.A \setminus S.R$. `allocInStore`(S, n, m) gets a new interval of locations on the store S, of length n and starting with a location who is multiple of m. That interval is added to $S.A$. `freeInStore`(S, l) deletes from $S.A \setminus S.R$ the interval of locations starting with location l. Notice that attempting to access a memory zone not included in $S.A$ leads to an abnormal termination, represented by the *error* result, except for `allocInStore` which returns the `NULL` location when memory is exhausted.

3.3 Environments

Each variable, array name and string constant in the C program is bound to the initial location of the interval in the C memory on which their data is represented. We call *environment* to that binding and define

$$\mathsf{Env_C} \triangleq \mathsf{Var_C} \uplus \mathsf{Array_C} \uplus \mathsf{String_C} \rightharpoonup \mathsf{Loc_C}$$

as the set of environments. There are two kinds of environments: the *global environment*, used for global variables, global arrays and all the byte string constants defined by the program; and *local environments* used for local variables and arrays defined locally in a function. The global environment is created by the `makeGlobalEnv` operation, which takes a C store and the sets of global objects declared in the program and returns the global environment and the C store with the memory zones allocated for those objects.

We also defined a special kind of environment which contains the necessary information for executing the functions called by the program. We define a *function environment* as a tuple $\langle fn, il, lr, L, sts \rangle$, where fn is the function name, il is the interval of locations that is allocated for local variables, lr is the initial location of the memory zone to which the result will be assigned, L is an environment whose domain is the set of local variables, and sts is the command stack containing the statements to be executed. Function environments are created by the `makeFuncEnv` operation, which takes a C store, a function name and a memory location where the return value will be stored. It returns the new function environment and a C store where the memory zones for actual parameters have been allocated. Fig. 3 shows a graphical view of a function environment, which can also be noted by fe.

3.4 Evaluation of expressions

We chose to specify different operations for evaluating expressions to abstract values as well as concrete values (sequences of bytes). Those operations take the name of the current function, an expression and the global and local environments. They return the requested value and a C store which must be equivalent to the store taken by the operation.

$$\begin{aligned}
\texttt{address} &: \mathsf{Function_C} \times \mathit{lexp} \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow (\mathsf{Store_C} \times \mathsf{Loc_C}) \uplus \{\mathit{error}\} \\
\texttt{evalSexpr} &: \mathsf{Function_C} \times \mathit{lexp} \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow (\mathsf{Store_C} \times \mathsf{Byte}^*) \uplus \{\mathit{error}\} \\
\texttt{evalIexpr} &: \mathsf{Function_C} \times \mathit{expr} \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow (\mathsf{Store_C} \times \mathsf{Int_C}) \uplus \{\mathit{error}\} \\
\texttt{evalLexpr} &: \mathsf{Function_C} \times \mathit{expr} \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow (\mathsf{Store_C} \times \mathsf{Loc_C}) \uplus \{\mathit{error}\}
\end{aligned}$$

`address` evaluates a "left expression" to a memory location. `evalSexpr` evaluates a left expression to a sequence of bytes representing a memory location. `evalIexpr` evaluates a normal expression which must be "typable" to a numeric type to an integer value. `evalLexpr` evaluates a normal expression which must be typable to a pointer type to a memory location value.

3.5 Assignment operations

The principle of our assignment operations is to update the C store with a sequence of bytes representing a value on a given memory location. The interval of locations computed from that memory location which has the same size of the sequence of bytes must be in the allocated memory intervals in the store.

$$\begin{aligned}
\texttt{assign} &: \mathsf{Function_C} \times \mathit{lexp} \times \mathit{expr} \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow \mathsf{Store_C} \uplus \{\mathit{error}\} \\
\texttt{assignRet} &: \mathsf{Function_C} \times \mathsf{Loc_C} \times \mathit{expr} \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow \mathsf{Store_C} \uplus \{\mathit{error}\} \\
\texttt{multAssign} &: \mathsf{Function_C} \times (\mathit{lexp} \times \mathit{expr})^* \times \mathsf{Env_C} \times \mathsf{Env_C} \times \mathsf{Store_C} \rightarrow \mathsf{Store_C} \uplus \{\mathit{error}\}
\end{aligned}$$

`assign` is the usual assignment operation. `assignRet` is the operation made at the return of a function, where the memory location has already been specified. Finally, the `multAssign` operation performs a "simultaneous" assignment for a sequence of expression couples $\langle le, e\rangle$, with an arbitrary order of evaluation.

3.6 Program configurations and transition rules

Now we can define with precision the configuration of a C program. Let $\mathsf{Conf_C}$ be the set of C configurations. A C configuration is a tuple $\langle \mathit{fes}, G, S, I, O\rangle$, where fes is a stack of function environments, G is the global environment for variables, array names and constant strings in the program, and I, O are respectively the input and output stream. A graphical view of a C configuration is shown in fig. 4.

Transition rules on C configurations [9] are defined with respect to the first statement in the command stack on the function environment at top of the function environment stack. We present two examples of transition rules with some simplifications for better readability. The following rule describes the execution of an assignation statement. It relies on the `assign` operation, which evaluates the left expression to a memory location and the right expression to a low level value as a sequence of bytes. The C store is updated on the memory zone beginning by the location with the evaluated value.

$$\frac{S_1 = \texttt{assign}(\mathit{fn}, \mathit{le}, e, G, L, S)}{\langle\langle \mathit{fn}, \mathit{il}, l, L, \langle \mathit{le} = e : \mathit{sts}\rangle\rangle : \mathit{fes}, G, S, I, O\rangle \longrightarrow \langle\langle \mathit{fn}, \mathit{il}, l, L, \mathit{sts}\rangle : \mathit{fes}, G, S_1, I, O\rangle}$$

The following rule describes the execution of a non void function call. The `address` operation gets the memory location associated to the left expression where the

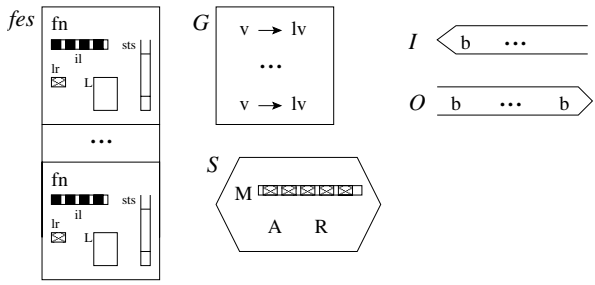

Fig. 4. View of a C configuration

return value will be stored. Then, a new function environment is created by the `makeFuncEnv` operation before actual parameters are evaluated and assigned to their respective arguments on the C store. That assignation is made by the `multAssign` operation.

$$\frac{\begin{array}{c}\langle S_1, l_1\rangle = \texttt{address}(fn, le, G, L, S) \\ \langle S_2, fe\rangle = \texttt{makeFuncEnv}(S, fn_1, l_1) \ \textit{where}\ fe = \langle fn_1, il_1, l_1, L_1, sts_1\rangle \\ S_3 = \texttt{multAssign}(fn_1, \langle (v_1, e1), \ldots, (v_n, e_n)\rangle, G, L_1, S_2) \ \textit{where}\ \langle v_1, \ldots, v_n\rangle = \mathit{ArgsFunc}(fn_1)\end{array}}{\langle\langle fn, il, l, L, \langle le = fn_1(e_1, \ldots, e_n) : sts\rangle\rangle : fes, G, S, I, O\rangle \ \longrightarrow\ \langle fe : \langle fn, il, l, L, sts\rangle : fes, G, S_3, I, O\rangle}$$

4 Building the C interface of a real programming language

In this section we present a concrete application of our framework to the formal specification of the C interface for a real, rule-based programming language.

4.1 The Russel programming language

Russel is the specific purpose language for the ASAX system [4], whose goal is the analysis of sequential data, such as computer logs and network traffic, in applications like auditing and network intrusion detection. Russel is a rule-based language with an imperative programming style. Concretely, a Russel program is a set of rules that perform some analysis over an *audit trail*. This audit trail is defined as a sequence of records, each record contains fields having a name and a value represented as a byte sequence. Russel has only two data types: integer and byte string. Programs define a set of global variables and rules, and each rule has its own local variables and parameters, with a corresponding action that is executed when an instance of the rule is triggered. Russel has a limited computing expressiveness, but current implementations provide the facility to call external functions written in C. This facility is defined in a relatively informal way.

Here is an example of a Russel program fragment containing a rule that counts the failed logins recorded on the audit trail where records have the fields `evt` and `res`. When the number of failed logins exceeds the number passed in the rule parameter, a global variable `toomany` is set to 1. Notice that this code does not use any call to a C function.

```
global toomany : integer;

rule failed_login(maxtimes : integer);
   newmaxtimes : integer;
begin
```

$$
\begin{array}{lll}
\textit{var\_decl} & ::= & \texttt{v} : \texttt{t} \\
\textit{rule} & ::= & \texttt{r}(\textit{var\_decl}*)\ \textit{var\_decl}*\ \textit{rule\_action} \\
\textit{rule\_action} & ::= & \texttt{skip} \\
 & | & \texttt{if}\ \textit{guarded\_action} + \ \texttt{fi} \\
 & | & \texttt{do}\ \textit{guarded\_action} + \ \texttt{od} \\
 & | & \textit{action}; \textit{action} \\
 & | & \texttt{trigger off}\ \textit{trigger\_mode}\ \texttt{r}(\textit{expression}*) \\
\textit{action} & ::= & \texttt{v}{:=}\textit{expression} \mid \textit{rule\_action} \\
\textit{guarded\_action} & ::= & \textit{condition} \rightarrow \textit{action} \\
\textit{trigger\_mode} & ::= & \texttt{for current} \mid \texttt{for next} \mid \texttt{at completion}
\end{array}
$$

Fig. 5. Russel EBNF syntax for rules

```
   if (maxtimes <= 0)
   -->   toomany := 1;
      (maxtimes > 0)
   -->   newmaxtimes := maxtimes;
         if (evt='login') and (res='failure')
         -->   newmaxtimes := maxtimes-1;
         fi
         trigger off for next failed_login(newmaxtimes);
   fi
end.
```

4.1.1 Syntax

Fig. 5 shows a fragment of the abstract syntax of Russel, in EBNF notation. In that syntax, `v` denotes a variable name, `t` a type name, and `r` a rule name. Expressions are defined as usual (variables, constant literals and arithmetic expressions), with a minor addition: valid field names of records in the audit trail are valid expressions, which are evaluated with respect to the current record. A condition is also added to usual ones (boolean literals, comparison and boolean operators) which checks whether a field name is defined in the current record.

4.1.2 Operational semantics

Informally, we can describe the execution of a Russel program as follows. The program reads the first record in the audit trail, and an initial action is executed. That initial action triggers rule instances to be effectively executed either for the current record, for the next record, or at the end of the audit trail. When all the actions are executed, the program selects randomly another triggered rule instance for the current record and executes its respective action. When all triggered rule instances for the current record are executed, the program reads the next record and takes the rule instances which have been triggered for that record. When all records in the trail have been read, the program takes the triggered instances for the end of the analysis.

We defined in [9] an abstract operational semantics for Russel. In that semantics, a program configuration is composed of: the audit trail, tr; a container for the rules which were triggered for the current record, cr, where each triggered rule instance contains values that will be assigned to local parameters; there are also two similar containers for the rules triggered for the next record and the end of analysis, named nr and cr respectively. In addition, a global environment G binds the global variables to their values, which can be integers or byte strings, and a

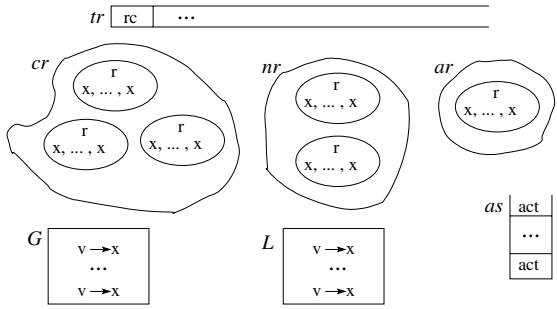

Fig. 6. View of a Russel configuration

local environment L binds values to the local variables of the rule instance that is being executed. Finally, a Russel configuration has an action stack, as, containing the actions to be executed for that instance. Fig. 6 shows a graphical view of a Russel configuration, which is denoted by $\langle cr, nr, ar, as, tr, G, L\rangle$.

Transition rules are defined with respect to the action at top of the action stack. For giving an example, we present the rule that triggers a rule instance for the next record. Notice the evaluation of parameters for the instance rule and their inclusion on the rule container for the next record.

$$\frac{x_1 = \mathit{EvalExpr}(e_1, tr, G_R, L_R) \quad \ldots \quad x_n = \mathit{EvalExpr}(e_n, tr, G_R, L_R)}{\begin{array}{l}\langle cr, nr, ar, \texttt{trigger off for next r}(e_1, \ldots, e_n) : as, tr, G_R, L_R\rangle \\ \langle cr, nr \cup \{\langle r, x_1, \ldots, x_n\rangle\}, ar, as, tr, G_R, L_R\rangle\end{array} \longrightarrow}$$

4.2 Extending the Russel syntax

We extend the Russel syntax by adding a new kind of action for calling functions which are actually implemented in C. In this extended syntax, fn is a C function name.

$$\begin{array}{lll} \mathit{annotated\_expr} & ::= & \mathit{expression} \mid \texttt{ref v} \\ \mathit{action} & ::= & \texttt{v:=}\mathit{expression} \mid \mathit{rule\_action} \mid \texttt{fn}(\mathit{annotated\_expr}*) \end{array}$$

Calls to C functions from Russel programs are made with *annotated expressions*, which indicate how the Russel data is effectively passed to the C program. Single Russel expressions are passed by value. An annotated expression `ref v` means that the Russel variable v is passed by result, that is, a Russel value will be assigned to v after the C function returns. The C programmer must ensure that a value respecting the type declaration of v is effectively built and put in the corresponding parameter, as we will precise in the next subsection.

4.3 Exchange data structure

We will define a mechanism for passing data from Russel to C based on partial memory copying. Russel values passed to C functions are represented in a specific interval of locations on the C store. In the C memory, Russel integers are represented as C integers, and Russel byte strings are represented by a pointer to an allocated memory zone whose content is composed of a sequence of `SIZEINT` bytes representing the length of the byte string, followed by the actual Russel sequence of bytes. Since output parameters are just Russel variables, their values are not passed at all, but C

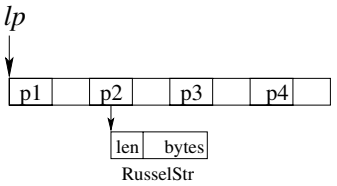

Fig. 7. Representation of Russel parameters in the C memory

programmers have the responsibility to fill those parameters with C values that can be effectively converted to Russel values, respecting the declared type of the Russel variable. Our exchange data structure is a tuple composed of the initial location of the allocated interval for Russel parameters and the annotated expressions that are passed to the C function. We show a graphical example in fig. 7, where parameter $p1$ is an integer, parameter $p2$ is a byte string, and the remaining parameters are passed by result.

4.4 Combining Russel and C semantics

The following operations perform the memory copying between Russel and C when a Russel program calls a C function:

$$\begin{array}{ll} \texttt{Reflection} & : \mathsf{AExpr_R}^* \times \mathsf{Store_C} \times \mathsf{Env_R} \times \mathsf{Env_R} \times \mathsf{Trail_R} \rightarrow (\mathsf{Store_C} \times \mathsf{Loc_C}) \uplus \{error\} \\ \texttt{Dereflection} & : \mathsf{AExpr_R}^* \times \mathsf{Loc_C} \times \mathsf{Store_C} \times \mathsf{Env_R} \times \mathsf{Env_R} \rightarrow (\mathsf{Store_C} \times \mathsf{Env_R} \times \mathsf{Env_R}) \uplus \{error\} \end{array}$$

$\texttt{Reflection}(Aes, S, G_R, L_R, tr)$ takes a sequence of annotated expressions Aes, a C store S, Russel environments G_R and L_R and the audit trait tr. It gives a C store S', such that $S' \succ_C S$ and S' has a new interval of locations allocated for the values of Aes, which are evaluated from the Russel environments G_R and L_R and the audit trail tr. Parameters passed by value are evaluated and assigned in their byte representation on their corresponding positions at the allocated C memory zone.

$\texttt{Dereflection}(Aes, l, S, G_R, L_R)$ takes a sequence of annotated expressions Aes, a location l, a C store S and Russel environments G_R and L_R. It gives a C store S', such that $S' \prec_C S$ and S' contains the allocated intervals of S without the location interval starting with l; and Russel environments G'_R and L'_R whose contents are the same of G_R and L_R except for output parameters whose values have been updated from the C memory. A combined Russel-C configuration is noted by

$$\langle\langle cr, nr, ar, as, tr, G_R, L_R\rangle, lp, Aes, \langle fes, G_C, S, I, O\rangle\rangle$$

Meta-rules given in definitions 2.1 to 2.4 are enough to give the transitions rules which preserve the original C and Russel semantics. For calling to and returning from C functions in Russel programs, we just need to add the following rules on combined Russel-C configurations:

$$(1)\quad \frac{\begin{array}{c}\langle S_0, fe\rangle = \texttt{makeFuncEnv}(S, \mathsf{fn}, \texttt{NULL}) \\ \langle S_1, lp\rangle = \texttt{Reflection}(Aes, S_0, G_R, L_R, tr)\end{array}}{\begin{array}{l}\langle\langle cr, nr, ar, \mathsf{fn}(Aes) : as, tr, G_R, L_R\rangle, \texttt{NULL}, \bot, \langle\bot, G_C, S, I, O\rangle\rangle \longrightarrow \\ \langle\langle cr, nr, ar, as, tr, G_R, L_R\rangle, lp, Aes, \langle[fe], G_C, S_1, I, O\rangle\rangle\end{array}}$$

```
void getRintParam(byte * Rparams, int n)
{  int * res;

   res = (int *) &(Rparams[n*SIZERPARAM]);
   return *res;
}
```

```
void setRintParam(byte * Rparams, int n, int x)
{  int * rpn;

   rpn = (int *) &(Rparams[n*SIZERPARAM]);
   *rpn = x;
}
```

```
void getRstrParam(byte * Rparams, int n)
{  int * res;

   res = (byte **) &(Rparams[n*SIZERPARAM]);
   return *res;
}
```

```
void setRstrParam(byte * Rparams, int n, byte * s)
{  int * rpn;

   rpn = (byte **) &(Rparams[n*SIZERPARAM]);
   *rpn = s;
}
```

Fig. 8. A simple API for the "ideal" Russel-C interface

$$(2)\qquad \frac{lp \neq \texttt{NULL} \qquad \langle S_1, G'_R, L'_R\rangle = \texttt{Dereflection}(Aes, S_0, G_R, L_R, tr)}{\langle\langle cr, nr, ar, as, tr, G_R, L_R\rangle, lp, Aes, \langle\bot, G_C, S, I, O\rangle\rangle \longrightarrow \langle\langle cr, nr, ar, as, tr, G'_R, L'_R\rangle, \texttt{NULL}, \bot, \langle\bot, G_C, S_1, I, O\rangle\rangle}$$

4.5 Examples of Russel-C programs

We can define a very simple API to be used by C programmers to write functions used in Russel programs. `getRintparam(lp,n)` gives the n-th parameter in the Russel parameters table as an integer. `getRstrparam(lp,n)` gives the n-th parameter in the Russel parameters table as an byte string. In a similar way, `setRintparam(lp,n,x)` and `setRstrparam(lp,n,s)` assigns an integer or byte string value to the n-th parameter in the Russel parameter table. We assume that `SIZERPARAM` represents the size of the byte representation for a Russel value in C, which we can determine from the actual size of C integers and pointers, The API is shown in fig. 8.

Here is a Russel rule which calls two C functions: `printS`, which just prints a string in the C output stream (notice that Russel does not have I/O primitives), and `stringReverse`, which generates a string with the contents of an input string in reverse order. The rule takes the `code` field of the current record, reverses it and print the reversed value before triggering another instance of itself for the next record.

```
rule reverse_code;
   icode : string;
begin
   stringReverse(code, ref icode);
   printS(icode);
   trigger off for next reverse_code
end.
```

The C implementation of those functions is presented below:

```
void stringReverse(byte * Rparams)
{  int numbytes, i, tmp;
   byte * Rstr;
   int * Rslen;

   Rstr = getRstrParam(Rparams, 0);
   Rslen = (int *) Rstr;
   numbytes = *Rslen;
   i = SIZEINT;
   while (i < numbytes/2){
      tmp = Rstr[nb+SIZEINT-i-1];
      Rstr[nb+SIZEINT-i-1] = Rstr[i];
      i = i+1;
   }
   setRstrParam(Rparams, 1, Rstr);
}
```

```
void printS(byte * Rparams)
{  byte * Rstr;
   int * Rslen;
   int numbytes, i;

   Rstr = getRstrParam(Rparams, 0);
   i = SIZEINT;
   Rslen = (int *) Rstr;
   numbytes = *Rslen;
   while (i < numbytes){
      print(Rstr[SIZEINT+i]);
      i = i+1;
   }
}
```

4.6 Towards a precise description of the real Russel-C interface

The combined operational semantics for Russel-C programs we presented is "ideal" since we considered the original Russel and C semantics as if they had independent implementations. To obtain a description of the real Russel-C interface we need to indicate some details of the actual implementation of Russel in C. It means that for an abstract Russel configuration $\langle cr, nr, ar, as, tr, G_R, L_R\rangle$, we need to specify how each one of its components is implemented in the C store. In fact, we refine the Russel configuration by adding "concrete" elements which give some implementation details. First, we define a *concrete Russel environment* as a partial function that binds Russel variables, record fields and Russel byte strings to locations in the C store:

$$\mathsf{CEnv_R} \triangleq \mathsf{Field_R} \uplus \mathsf{Var_R} \uplus \mathsf{String_R} \rightharpoonup \mathsf{Loc_C}$$

Second, we define a concrete Russel configuration as a tuple

$$\langle Conf_R, \mathsf{r}, Cf, ilf, Cg, ilg, Cl, ill, ils, S_C\rangle$$

where $Conf_R$ is an abstract Russel configuration, r is the name of the current rule instance, Cf and ilf are respectively the concrete Russel environment and the allocated interval of locations on the C store for the current record of the audit trail. Cg and ilg are the concrete environment and allocated interval for Russel global variables. Cl and ill are similarly defined for local variables. ils is an interval of locations on the C store that is allocated for storing the byte strings that are passed as parameters in the current rule instance. S_C is the C store shared by the combined Russel-C program. We add operations for creating concrete Russel environments and for evaluating Russel expressions on those concrete environments.

Afterwards, we refine the transition rules of the abstract semantics of Russel to describe the changes on concrete elements of the refined configuration. This is straighforward for transition rules which do not modify the Russel environments, and for the other rules we need to specify how concrete environments and allocated memory are modified. Finally, the `Reflexion` and `Dereflexion` operations are redefined to have in account the fact that Russel data can be found directly in the C store. Since we keep the abstract Russel configuration in the concrete configuration, we need to specify a coherence condition stating that Russel values in abstract environments are well implemented in the C store for concrete environments. This must be ensured by the `Dereflexion` operation.

Transition rules for calling a C function from Russel and returning from a C function to Russel show explicitly the changes on the concrete environments:

$$(3)\quad \frac{\begin{array}{c}\langle S_0, fe\rangle = \texttt{makeFuncEnv}(S_C, \mathsf{fn}, \texttt{NULL})\\ \langle S_1, lp\rangle = \texttt{Reflection}(Aes, S_0, \mathsf{r}, G_R, Cg, L_R, Cl, tr, Cf)\end{array}}{\begin{array}{l}\langle\langle cr, nr, ar, \mathsf{fn}(Aes) : as, tr, G_R, L_R\rangle, \mathsf{r}, Cf, ilf, Cg, ilg, Cl, ill, ils, S_C\rangle, \texttt{NULL}, \bot, \langle\bot, G_C, S_C, I, O\rangle\rangle \longrightarrow\\ \langle\langle cr, nr, ar, as, tr, G_R, L_R\rangle, \mathsf{r}, Cf, ilf, Cg, ilg, Cl, ill, ils, S_1\rangle, lp, Aes, \langle[fe], G_C, S_1, I, O\rangle\rangle\end{array}}$$

$$(4)\quad \frac{\begin{array}{c}lp \neq \texttt{NULL}\\ \langle S_0, G'_R, L'_R\rangle = \texttt{Dereflection}(Aes, S_C, \mathsf{r}, ilf, Cg, ilg, Cl, ill)\\ S1 = \texttt{freeInStore}(S_0, lp)\end{array}}{\begin{array}{l}\langle\langle cr, nr, ar, as, tr, G_R, L_R\rangle, \mathsf{r}, Cf, ilf, Cg, ilg, Cl, ill, ils, S_C\rangle, lp, Aes, \langle\bot, G_C, S_C, I, O\rangle\rangle \longrightarrow\\ \langle\langle cr, nr, ar, as, tr, G'_R, L'_R\rangle, \mathsf{r}, Cf, ilf, Cg, ilg, Cl, ill, ils, S_1\rangle, lp, Aes, \langle\bot, G_C, S_1, I, O\rangle\rangle\end{array}}$$

All the details of the real Russel-C interface are given in [9].

5 Further work and conclusions

Whereas C interfaces in programming language implementations have been used in practice, there are not many efforts to formally describe those FLI. They are mostly left out of the formal description of programming languages. In this paper, we show that it is possible to describe a C interface by the application of a conceptual framework based on structural operational semantics. This formal description allows us to verify the potential side effects caused by calling C code in programs that use the C interface. Such side effects include memory leaks and corruption of the program configuration.

Another framework for describing the execution of multi-language programs based on operational semantics is given by Matthews and Findler in [7]. Their formalism for operational semantics is lambda calculus, with "embeddings" which model inter-language interactions.

Our C semantics is different from other developed semantics for that language. Blazy and Leroy developed a memory model and a big-step operational semantics for C in [3]. Our memory model is at a lower level than theirs, since we define a memory as a function from locations to bytes instead of using abstract values. Nevertheless, we can specify how abstract values are built from their byte representation. Another formal semantics was given by Papaspyrou in his Ph.D. thesis [11], which is a denotational semantics. We use an operational semantics framework since we found it more natural to describe the interoperability issues of language interfaces. An interesting direction for further work is the validation of our semantics with formal tools, as it is done by Norrish in his Ph.D. thesis [8]. He developed a big-step operational semantics for C which has been validated in HOL.

In the short term we want to apply our methodology to a formal description of C interfaces with more complex languages, like the Java Native Interface [6], and we will also explore the multiparadigm aspects of interoperability in declarative logic languages like Prolog, which have several C interfaces [2].

References

[1] C programming language standard ISO/IEC 9899. http://www.open-std.org/JTC1/SC22/WG14/.

[2] Bagnara, R., and Carro, M., "Foreign language interfaces for Prolog: a terse survey," Quaderno 283, Dipartimento di Matematica, Università di Parma, Italy, 2002. http://www.cs.unipr.it/~bagnara.

[3] Blazy, S., and Leroy, X., "Formal verification of a memory model for C-like imperative languages," In *Proceedings of ICFEM'05*, pages 280-299, Manchester, UK, 2005.

[4] Habra, N., Le Charlier, B., Mounji, A., and Mathieu, I., "ASAX. software architecture and rule-based language to universal audit trail analysis," In *European Symposium on Research in Computer Security (ESORICS)*, pages 435-450, 1992.

[5] Kernighan, B. W., and Ritchie, D. M., "The C programming language", 2nd Edition, Addison-Wesley, 1988.

[6] Liang, S., "The Java Native Interface: Programmer's Guide and Specification", Addison-Wesley, 1999.

[7] Matthews, J., and Findler, R. B., "Operational semantics for multi-language programs," In *Proceedings of POPL'07*, Nice, France, 2007.

[8] Norrish, M., "C formalized in HOL," Ph.D. thesis, Computer Laboratory, University of Cambridge, U.K., 1998.

[9] Ospina, G. A., "Un Cadre Conceptuel pour l'Interopérabilité des Langages de Programmation," Ph.D. thesis, Université catholique de Louvain, Belgium, 2007.

[10] Ospina, G. A., and Le Charlier, B., "Towards precise descriptions for programming language interoperability: a general approach based on operational semantics," In *Proceedings of I-ESA'07*, Madeira, Portugal, 2007.

[11] Papaspyrou, N. S., "A Formal Semantics for the C Programming Language," Ph.D. thesis, National Technical University of Athens, Greece, 1998.