1. The Restricted C Fragment: Xd-C

The restricted C fragment called Xd-C is confined in a subset of ANSI-C (C89 standard version). It consists of often-used types, expressions and statements of C language. Xd-C is similar to Clight [1] but more than Clight.

1.1. Types

The supported types in Xd-C include arithmetic types (char, int, float and double in various sizes and signedness), pointer, void pointer, function pointer and struct types. However, union type, static local variables and type qualifiers such as const, restrict and volatile are not allowed in Xd-C. As storage-class specifiers, typedef definitions have been expanded away during parsing and type-checking. The abstract syntax of Xd-C types is given as follows:

$$
\begin{array}{lrl}
\text{Signedness} & sign & ::= signed \mid unsigned \\
\text{int length} & len & ::= short \mid long \\
\text{Types} & \tau & ::= int \mid sign\ int \mid len\ int \mid sign\ len\ int \\
 & & \quad \mid float \mid double \mid long\ double \mid char \mid sign\ char \\
 & & \quad \mid structself \mid voidp \mid functp \mid \tau *
\end{array}
$$

Self-defined Types:

$$
\begin{array}{ll}
structself & ::= struct\ id_1\{(\tau\ id_2;)^+\} | struct\ id_1 \\
functp & ::= [\tau|void]((\tau,)^*\tau)* \mid [\tau|void]()* \\
voidp & ::= void*
\end{array}
$$

where $struct\ id_1\{(\tau\ id_2;)^+\}$ defines a structure id_1 consisting of body $(\tau\ id_2;)^+$; $[\tau|void]((\tau,)^*\tau)*$ defines a function pointer with each parameter in type τ and a return value in type τ or $void$. $[\tau|void]()*$ defines a function pointer with no parameter. Note that id (possibly with subscriptions) is a string (name) consisting of characters and digits with a character as its head.

1.2. Expressions

The expression e in Xd-C is inductively defined as follows:

$$
\begin{array}{rlrl}
e ::= & \multicolumn{3}{l}{c \mid x \mid x(e_1, ..., e_m) \mid (\tau)\ e \mid op_1\ e \mid e_1\ op_2\ e_2 \mid e_1?e_2 : e_3 \mid (e)} \\
x ::= & \multicolumn{3}{l}{id \mid id[e] \mid id[e_1][e_2] \mid e.x \mid e \rightarrow x} \\
op_1 ::= & \& \mid * \mid + \mid - \mid \tilde{} \mid\ ! & op_2 ::= & aop \mid bop \mid rop \mid eop \mid lop \\
aop ::= & + \mid - \mid * \mid / \mid \% & bop ::= & << | >> | \ \& \mid | \mid \hat{} \\
rop ::= & < | > | <= | >= & eop ::= & == | != \\
lop ::= & \&\& \mid || & &
\end{array}
$$

where c is an arbitrary constant, id a variable, $id[e]$ the eth element of array id (counting from 0), $id[e_1][e_2]$ the element in row e_1 and column e_2 , $e.x$ the member x of structural variable e, $e \to x$ the member x of the structural variable that e points to and $x(e_1, ..., e_m)$ a function call with arguments $e_1, ..., e_m$. $(\tau)\ e$ represents the type cast of e namely converting the value of e to the value with type τ. $op_1\ e$ is a unary expression including $\&e$ taking the address of e, $*e$ the pointer dereferencing, $+e$ the positive of e, $-e$ the negative of e, $\tilde{}e$ the bitwise complement of e and $!e$ the logical negation of e. op_2 represents an binary operator including arithmetic operators aop ($+$, $-$, $*$, / and %), bitwise operators bop ($<<$, $>>$, &, | and ˆ), relational operators rop ($<$, $>$, $<=$ and $>=$), equality operators eop ($==$ and $!=$) and logical operators lop (&& and ||). $e_1?e_2 : e_3$ is a conditional expression indicating that the result is e_2 if e_1 is not equal to 0, and e_3 otherwise.

The expression le that can occur in left-value position can inductively defined as follows:

$$le ::= x | *e$$

1.3. Statements

The following are the elementary statements in Xd-C:

Statements:	$s ::=;$	null
	$\|e;$	expression
	$\|le = e;$	simple assignment
	$\|$`if(`e`)`$\{s_1\}$`else`$\{s_2\}$	conditional
	$\|$`switch(`e`)`$\{sw\}$	switch
	$\|$`while(`e`)`$\{s\}$	while loop
	$\|$`do`$\{s\}$`while(`e`);`	do loop
	$\|$`for(`$s_1; e; s_2$`)`$\{s\}$	for loop
	$\|$`continue;`	next iteration of the current loop
	$\|$`break;`	exit from the current loop
	$\|$`return` e;	return from the current function
	$\|$`return;`	
	$\|s_1; s_2$	sequence
Switch cases:	$sw ::= sw_1 \| sw_2; sw_1$	
	$sw_1 ::=$ `default :` $s;$	default case
	$sw_2 ::=$ `case` $n :\ s; sw_2$	labeled case

A null statement performs no operations. An expression statement (e;) is evaluated as a void expression for its side effects. In simple assignment statement ($le = e$;), the value of e replaces the value stored in the location designated by le. In conditional statement `if(`e`)`$\{s_1\}$`else`$\{s_2\}$, s_1 executes if the expression e compares unequal to 0, s_2 executes otherwise. In switch statement `switch(`e`)`$\{sw\}$, e is the controlling expression, the expression of each case label shall be an integer constant expression. There are three kinds of iteration statements in Xd-C including while loop, do loop and for loop statements. An iteration statement causes the loop body to execute repeatedly until the controlling expression equals 0. In while loop statement `while(`e`)`$\{s\}$, the evaluation of e takes place before each execution of s, while in do loop statement `do`$\{s\}$`while(`e`)`;, the evaluation of e takes place after each execution of s. In for loop statement `for(`$s_1; e; s_2$`)`$\{s\}$, s_1 executes once at the beginning of the first iteration, e is the loop condition, s_2 executes at the end of each iteration, and s is the loop body. Jump statements including `continue`;, `break`;, `return` e; and `return`; are supported in Xd-C, but not the `goto` statement. A `continue` statement shall appear only in or as a loop body. A `break` statement terminates execution of the smallest enclosing `switch` or iteration statement. A `return` statement appears only in a function.

A Xd-C program is composed of a list of declarations, a list of functions and a main function. It can be defined as follows:

Array	$array$	$::=$	$\tau\ id[n] \mid \tau\ id[m][n] \mid \tau\ id[n] = \{(e,)^*e\}$
			$\mid \tau\ id[] = \{(e,)^*e\} \mid \tau\ id[m][n] = \{(e,)^*e\}$
			$\mid \tau\ id[m][n] = \{(\{(e,)^*e\},)^*\{(e,)^*e\}\}$
Structure	$structure$	$::=$	$struct\ id_1\{(\tau\ id_2;)^+\}$
Variable list	$varlist$	$::=$	$id \mid id = e \mid varlist, varlist$
Declaration	Pd	$::=$	$\tau\ varlist \mid array \mid structure$
Function	$funct$	$::=$	$[\tau \vert void]\ id_1((\tau\ id_2,)^*(\tau\ id_2))\{(Pd;)^*s\}$
			$\mid [\tau \vert void]\ id_1()\{(Pd;)^*s\}$
Program	P	$::=$	$(Pd;)^*(funct;)^*$
			$int\ main(int\ argc, char{*}{*}\ argv)\{(Pd;)^*s\}$

where $\tau\ id[n]$ defines a one dimensional array id having n elements with type τ while $\tau\ id[m][n]$ defines a two dimensional array id having $m \times n$ elements with type τ; $id = e$ defines an initialization of id except for $structself$; $[\tau|void]\ id_1((\tau\ id_2,)^*(\tau\ id_2))\{(Pd;)^*s\}$ defines a function id_1 with each parameter id_2 in type τ and a return value in type τ or $void$; $[\tau|void]\ id_1()\{(Pd;)^*s\}$ defines a function id_1 with no parameter.

Summary: As we can see, some constructs and facilities in ANSI-C (C89) are not supported in Xd-C. In the following, we show a key negative list which Xd-C does not support.

(1) `goto` statement;

(2) $union$ structure;

(3) $e{+}{+}$, $e{-}{-}$, ${+}{+}e$ and ${-}{-}e$ expressions;

(4) $(a = b, b = c, d = (f(x), 0))$ comma statements;

(5) $op =$ compound assignments where $op ::= + \mid - \mid * \mid / \mid \% \mid >> \mid << \mid \& \mid$ $\mid \mid \hat{}$;

(6) $struct\ A\ a$; $a = \{(void()*)b, (void()*)c\}$ structure assignments;

(7) $x = y = z$ continuous assignments;

(8) $typedef$, $extern$, $static$, $auto$ and $register$ storage-class specifiers;

(9) $const$ and $volatile$ type qualifiers;

(10) local variables in a block;

(11) nested cases in a switch statement;

(12) assignment expressions such as $if((y = fun()) == x)$;

(13) function pointers pointing to external functions;

(14) functions that accept a variable number of arguments.

In fact, the constructs and facilities in the above negative list except for `goto` statement can be implemented by Xd-C although the implementation might be tedious. Therefore, Xd-C is a reasonable subset of ANSI-C (C89) in practice.

References

[1] S. Blazy, X. Leroy, Mechanized semantics for the clight subset of the c language, Journal of Automated Reasoning 43 (3) (2009) 263–288.