Converting C to LCTRSs

Naoki Nishida and Cynthia Kop

December 24, 2015

Abstract

This manuscript introduces a conversion of programs in a subclass of C to logically constrained term rewriting systems (LCTRS) [1]. We first show the syntax of the subclass, and then show a conversion of programs in the subclass to LCTRSs.

1 Syntax

Figure 1 illustrates the syntax of a subclass of C programs which we convert to LCTRSs. We may omit parentheses in a usual way. Note that function calls fit *two* syntaxes: they could be either expressions or direct statements – the former is intended for *integer* functions, the latter for both *void* and *integer* functions.

We restrict programs obtained by the above syntax to those which are successfully compiled as C programs without any warning; for example, it is not permitted to initialise a global variable to anything but a constant. It is also technically required that all variable declarations in a function occur at the start of the body, but we will not need this limitation. The semantics of programs generated by the above syntax follows the standard one of C programs.

2 Preparsing

Before translating the program, we take the following preparsing steps:

1. every declaration in the program, both globally and as a statement in functions, is split up into single declarations; that is, any occurrence of $T\ a_1, \ldots, a_n\,\texttt{;}$ with $n > 0$ is replaced by $T\ a_1\,\texttt{;} \ldots T\ a_n$;
2. every *argument* $T\ \texttt{x}[]$ and $T\ *\texttt{x}$ is replaced by $T*\ \texttt{x}$ (as this will allow us simply to write array arguments in the form $C\ \texttt{x}$ where $C \in \{\texttt{int}, \texttt{int}*\}$);
3. global variable declarations are normalised:
   (a) move all variable declarations without initialisation (i.e. $T\ \texttt{x;}$ or $T\ \texttt{x}[n]\texttt{;}$) to the start of the file, and move all variable declarations with initialisations directly behind (this is compiler-safe, as initialisations for global variables are not allowed to include function calls);
   (b) if any global variable is declared twice (this is permitted, as long as the variable is only *initialised* once, since the other occurrences count as prototypes), remove the duplicates, as follows:

(program)	P	::=	$D \ \ldots \ D$	
(definition)	D	::=	$ResT\ f\texttt{(}Arg\texttt{, } \ldots \texttt{, } Arg\texttt{)\{ }Ss\texttt{ \}}$	(declaration of function)
		|	$ResT\ f\texttt{(}Arg\texttt{, } \ldots \texttt{, } Arg\texttt{);}$	(prototype of function)
		|	$Decl\texttt{;}$	(declaration of global variable)
(resulting type)	$ResT$	::=	$T \mid \texttt{void}$	
(base type)	T	::=	`int`	
(argument)	Arg	::=	$T\ x \mid T\ x[] \mid T\ *x$	
(declaration)	$Decl$	::=	$T\ Inits$	(declaration of local variables)
(initializations)	$Inits$	::=	$Init \mid Init, Inits$	
(initialization)	$Init$	::=	$y \mid y[n] \mid y\texttt{ = }E \mid y[n]\texttt{ = }\{E_1, \ldots, E_k\}$	(with $k \leq n$)
(statements)	Ss	::=	$\epsilon \mid S\ Ss$	
(statement)	S	::=	$E\texttt{;}$	(expression)
		|	$Decl\texttt{;}$	(declaration)
		|	$f\texttt{(}E, \ldots, E\texttt{);}$	(function call)
		|	$\texttt{if(}\ E\ \texttt{)}\ S$	(`if`)
		|	$\texttt{if(}\ E\ \texttt{)}\ S\ \texttt{else}\ S$	(`if`)
		|	$\texttt{while(}\ E\ \texttt{)}\ S$	(`while`)
		|	$\texttt{do\{}\ Ss\ \texttt{\}}\ \texttt{while(}\ E\ \texttt{);}$	(`do-while`)
		|	$\texttt{for(}\ E\ \texttt{;}\ E\ \texttt{;}\ E\ \texttt{)}\ S$	(`for`)
		|	$\texttt{\{}\ Ss\ \texttt{\}}$	(block)
		|	`continue;`	
		|	`break;`	
		|	$\texttt{return}\ E\texttt{;}$	
		|	`return;`	
(expression)	E	::=	$n \mid Strg \mid \texttt{(-}\ E\texttt{)} \mid \texttt{(+}\ E\texttt{)} \mid E\texttt{, }E$	
		|	$\texttt{(}E\ Op\ E\texttt{)} \mid f\texttt{(}E\texttt{, } \ldots \texttt{, } E\texttt{)}$	
		|	$A \mid \texttt{(!}\ E\texttt{)} \mid \texttt{(}E\ \texttt{\&\&}\ E\texttt{)} \mid \texttt{(}E\ \texttt{||}\ E\texttt{)}$	
		|	$\texttt{(}E\ \texttt{?}\ E\ \texttt{:}\ E\texttt{)}$	
(assignment)	A	::=	$\texttt{(}Strg\texttt{ = }E\texttt{)} \mid \texttt{(++ }Strg\texttt{)} \mid \texttt{(--}\ Strg\texttt{)}$	
		|	$\texttt{(}Strg\texttt{ ++)} \mid \texttt{(}Strg\texttt{ --)} \mid \texttt{(}Strg\texttt{ += }E\texttt{)}$	
		|	$\texttt{(}Strg\texttt{ -= }E\texttt{)} \mid \texttt{(}Strg\texttt{ *= }E\texttt{)}$	
		|	$\texttt{(}Strg\texttt{ /= }E\texttt{)} \mid \texttt{(}Strg\texttt{ \%= }E\texttt{)}$	
(storage)	$Strg$	::=	$x \mid x[E]$	
(binary operators)	Op	::=	`+` | `-` | `*` | `/` | `%` | `==` | `!=` | `<` | `>` | `<=` | `>=`	

where

- f, $x_1, \ldots, x_n$, $y_1, \ldots, y_m$, x are identifiers, and
- n is an integer.

Figure 1: Syntax of C programs to be converted here

i. if the first has the form T `x` or T `x[]`, remove it (since it must have the same type as the second, as the compiler would protest otherwise, and cannot give any more information)

ii. if the first has the form T `x`$[n]$ and the second has the form T `x[]`, remove the latter;

iii. if the first has the form T `x`$[n]$ and the second has the form T `x[]` $= \{k_1, \ldots, k_m\}$, remove the former and replace the latter by T `x`$[n] = \{k_1, \ldots, k_m\}$; if the latter already has this form, just remove the former

note that these are the only forms the declarations can have: the first *cannot* be an initialised variable declaration, since we put all non-initialised declarations at the start of the program in step 3a, and the compiler should protest on double initialisations (if this happens anyway, we can throw an error and abort);

4. inside every function, all variable names are made unique and not overlapping with global variable names, and types between functions are consistent:

   (a) if a variable is declared (or used as a parameter) which was declared globally or earlier in the same function, then rename it and all subsequent occurrences in the same scope to a fresh name;

   (b) if a variable is declared (or used as a parameter) in one function with a type C (where a declaration `int x`$[n]$ counts as a type `int`$*$), when it was declared with a different type in another function, then rename the second variable and all its occurrences in the same function

5. every prototype $ResT\ f(a_1, \ldots, a_n)$`;` is removed, since these only serve for correctness checks which we assume done as the program compiles;

6. to have fewer cases to consider, we also make the following changes to assignments occurring in expressions anywhere in the program:

   (a) `S` $\circ$`= e` with $\circ \in \{+, -, *, /, \%\}$ is replaced by `S = S` $\circ$ `e`;

   (b) `++S` and `--S` are replaced by `S = S + 1` and `S = S - 1` respectively;

   (c) `S++` and `S--` are replaced by `S = S + 1, S - 1` and `S = S - 1, S + 1` respectively.

7. finally, for the same purpose, we change composite statements, replacing:

   (a) `if(` E `) {` Ss `}` by `if(` E `) {` Ss `} else {` ϵ `}`

   (b) `if(` E `)` S by `if(` E `) {` $S\epsilon$ `} else {` ϵ `}` if S is not a block

   (c) `if(` E `)` S_1 `else` S_2 by `if(` E `)` S_1' `else` S_2' where each $S_i' = S_i$ if S_i is a block, otherwise $S_i' =$ `{` $S_i\epsilon$ `}`

   (d) `while(` E `)` S by `while(` E `) {` $S\epsilon$ `}` if S is not a block

   (e) `for(` E_1 `;` E_2 `;` E_3 `)` S by `for(` E_1 `;` E_2 `;` E_3 `) {` $S\epsilon$ `}` if S is not a block

   so afterwards the statements of all `if` and `else` statements and all loops can be assumed to be blocks, and every `if` has a matching `else`.

It is clear that these changes do not affect the program's semantics. Afterwards, all variables are uniquely defined by their name and the function they occur in, global variables are all collected together at the start of the program, and the only remaining assignments have the form `S = e`.

In addition, because our transformation to LCTRSs only functions when different array variables necessarily refer to different arrays, we make sure that no overlaps occur:

8. if there is any function call $\mathtt{f}(e_1, \ldots, e_n)$ in the program where e_i is a global array variable $\mathtt{z}$ for some i, then:
   - let $\mathtt{f}'$ be a fresh function symbol;
   - copy the function declaration, taking into account the global variable: if $\mathtt{f}$ is declared as $\mathit{ResT}\ \mathtt{f}(C\ \mathtt{x}_1, \ldots, C\ \mathtt{x}_n)\{\mathit{Ss}\}$, then add the function $\mathit{ResT}\ \mathtt{f}'(C\ \mathtt{x}_1, \ldots, \mathtt{x}_{i-1}, \mathtt{x}_{i+1}, \ldots, \mathtt{x}_n)\{\mathit{Ss}[\mathtt{x}_i := \mathtt{z}]\}$ to the bottom of the program;
   - replace all function calls $\mathtt{f}(a_1, \ldots, a_n)$ in the program where $a_i = \mathtt{z}$ by $\mathtt{f}'(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$.

   Repeat this until no such function calls remain;
9. for all function calls $\mathtt{f}(e_1, \ldots, e_n)$ in the program where $i < j$ exist such that $e_i = e_j$ is an array variable:
   - let $\mathtt{f}'$ be a fresh function symbol;
   - copy the function declaration, taking into account the duplicate: if $\mathtt{f}$ is declared as $\mathit{ResT}\ \mathtt{f}(C\ \mathtt{x}_1, \ldots, C\ \mathtt{x}_n)\{\mathit{Ss}\}$, then add the function $\mathit{ResT}\ \mathtt{f}'(C\ \mathtt{x}_1, \ldots, \mathtt{x}_{j-1}, \mathtt{x}_{j+1}, \ldots, \mathtt{x}_n)\{\mathit{Ss}[\mathtt{x}_j := \mathtt{x}_i]\}$ to the bottom of the program;
   - replace all occurrences of $\mathtt{f}(a_1, \ldots, a_n)$ in the program where $a_i = a_j$ by $\mathtt{f}'(a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n)$.

   Repeat this until no such function calls remain;
10. If any functions are not called at all in the program, and are not required for analysis, remove their declarations.

3 Data Gathering

First, we save all global variable names in the mapping globals, which maps each variable to a unique index (for which we can simply choose the number of variables declared earlier in the program). We also save a set arrayvars containing those global variables declared as arrays, and a mapping asizes mapping arrays to their sizes and initials mapping variables to the value they are initialised with. This is easily done by a quick iteration over the declarations part of the program. Note that we do not have to evaluate expressions to calculate initials, since global variables must be initialised with constants. When an array variable is declared as $T\ \mathtt{x}[]$ no size is saved (it could have any size, depending on the compiler!) and $T\ \mathtt{x}[] = \{k_1, \ldots, k_n\}$ is saved to have size n.

Next, we define functions reads, writes, each mapping function symbols f to a set of variable names, and alters, which maps function symbols to a set of indexes, as the smallest sets satisfying the following property:

- if a variable x in globals is read in f (this includes array selections), then $x \in \mathsf{reads}(f)$;
- if a variable x in globals is written to in f (this includes array stores), then $x \in \mathsf{reads}(f)$ and $x \in \mathsf{writes}(f)$;
- if the i^{th} parameter to f is an array argument which is altered in f using array stores, then $i \in \mathsf{alters}(f)$;
- if a function call $g(e_1, \ldots, e_n)$ appears in the definition of f, then:
  - $\mathsf{reads}(g) \subseteq \mathsf{reads}(f)$;
  - $\mathsf{writes}(g) \subseteq \mathsf{writes}(f)$;
  - if e_i is an array variable `z` which is the j^{th} parameter to f, and $i \in \mathsf{alters}(g)$, then $j \in \mathsf{alters}(f)$.

Furthermore, we define the set $\mathcal{S} = \{\texttt{Int}, \texttt{IntArray}\} \cup \{\texttt{result}_\texttt{f} \mid f \text{ a function declared in the program}\}$; this is the set of sorts that we will use. We let $\mathcal{V}$, the set of variables in the LCTRS we use, contain all variables in the program, along with a sort. Here, sorts are assigned as follows:

- if `x` is a global variable in arrayvars, then $\mathit{sort}(\texttt{x}) = \texttt{IntArray}$;
- if `x` is a global variable not in arrayvars, then $\mathit{sort}(\texttt{x}) = \texttt{Int}$;
- if `x` is passed to any function as a variable `int* x`, or if there is a `int x` or `int x = e` in any function, then $\mathit{sort}(\texttt{x}) = \texttt{Int}$;
- in all other cases, `x` can only be locally declared or passed as an array variable, so $\mathit{sort}(\texttt{x}) = \texttt{IntArray}$.

4 Undefined Behaviour

One of the complications we must deal with is *unspecified* behaviour: the C-standard allows function calls, assignments, array lookups (which may potentially lead to faults) and so on to be done in any order between two sequence points. For example, it is perfectly allowed to write down an expression `(x = 3) + (x = 5)`, which could return each of 6, 8 and 10, and result in `x` being either 3 or 5. Similarly, a statement `x = x++;` with `x` initially set to 1 could result in `x` being either 1 or 2. The problem, in both cases, is that `x` is assigned to twice. Similarly, the result of `(x = 3) + x` is unspecified because `x` is both written to, and read in a separate part of the expression.

For simplicity, we will not accept programs which have such unspecified behaviour. That is, for every expression `e` in the program we test whether it is fully specified, using the functions *reading*, *writing*, *assigning*, and *specified*:

- *reading*(*exp*) contains every variable (normal or array, global or local) `x` such that one or more of the following holds:

  - `x` occurs directly in *exp*
  - `x` $\in$ `reads(f)` for some symbol `f` which occurs in e

- *assigning*(*exp*) contains every variable (normal or array, global or local) `x` such that either `x =` e or `x[`e_1`] =` e_2 appears in *exp*;
- *writing*(*exp*) contains every variable (normal or array, global or local) `x` such that `x` $\in$ *assigning*(*exp*) or *exp* has a sub-expression `f(`e_1`,...,`e_n`)` with either `x` $\in$ `writes(f)` or e_i = `x` for some $i \in$ `alters(f)`;
- *specified*(*exp*) is defined recursively over *exp*, as follows:
  - *specified*(n) = *specified*(`x`) = $\top$;
  - *specified*(`x[`e`]`) = *specified*(`-`e) = *specified*(`+`e) = *specified*(`!`e) = *specified*(e);
  - *specified*(e_1`,`e_2) = *specified*(e_1`&&`e_2) = *specified*(e_1`||`e_2) = *specified*(e_1)$\wedge$ *specified*(e_2) (since here, evaluation order is defined left-to-right);
  - *specified*(e_1 `?` e_2 `:` e_3) = *specified*(e_1) $\wedge$ *specified*(e_2) $\wedge$ *specified*(e_3) (here, too, evaluation order is defined left-to-right);
  - *specified*($e_1 \circ e_2$) = *specified*(e_1)$\wedge$*specified*(e_2)$\wedge$*reading*(e_1)$\cap$*writing*(e_2) = $\emptyset \wedge$ *writing*(e_1) $\cap$ *reading*(e_2) = $\emptyset$ where $\circ \in \{$`+`, `-`, `*`, `/`, `%`$\}$;
  - *specified*(`f(`e_1`,...,`e_n`)`) = *specified*(e_1) $\wedge \cdots \wedge$ *specified*(e_n) $\wedge \forall i, j \in \{1, \ldots, n\}[i \neq j \Rightarrow$ *reading*(e_i) $\cap$ *writing*(e_j) = $\emptyset]$;
  - *specified*(`x =` e) = *specified*(e) $\wedge$ `x` $\notin$ *assigning*(e);
  - *specified*(`x[`e_1`] =` e_2) = *specified*(e_1)$\wedge$*specified*(e_2)$\wedge$`x` $\notin$ *assigning*(e_1)$\wedge$ `x` $\notin$ *assigning*(e_2).

  Note that we *do* allow `x = f()` when `f` might modify `x`, because `f` has to be fully evaluated before its value can be assigned to `x`; thus, the assignment to `x` is always later. We do *not* allow `x = 3 + (x = 1)` since it is possible that the assignment `x = 1` is evaluated only after the assignment `x = 4`.

Clearly, testing whether all (outermost) expressions in the program are fully specified gives false negatives. For example, `(x = 1) + (x = 1)` is rejected even though there is only one reasonable outcome. Also, more problematically, if `f(a,i)` manipulates argument `i` of array `a`, then `f(a,1) + f(a,2)` is wrongfully rejected (as the same array is manipulated in multiple ways). A more general solution would be to allow all expressions and simply encode the nondeterminism in the resulting LCTRS, but for now we have elected not to do so.

Thus, we assume a preparsing step is done which rejects all programs containing expressions e for which *specified*(e) returns $\bot$. This rejection could either lead to a failure, or just a warning; from the point of view of grading student programs, this is where a human should look and manually assert that no evil is done.

Henceforth, we will therefore assume such expressions either do not occur in the program, or if they do, that it is acceptable to fix an evaluation order, which may lead to the conclusion that the program is correct even when this does not hold for all compilers.

5 Expressions and Side Effects

In this section, we will define a function to bring expressions into a more appealing form, where all the side effects happen at the start of an expression, separated by commas from the result.

To this end, we define a *pure expression* as an expression without any assignments or function calls. Ideally, we would like to split a given C-expression into a sequence of assignments and function calls and a pure expression. For example, $(x = 4) + y$ would become $x = 4, x + y$.

5.1 Consideration: Omitting Evaluations

A difficulty to take into account is that certain operators do not need to be fully defined: the evaluation of the left-most part determines the evaluation of the rest. For example, in `a && b`, if `a` evaluates to 0 then `b` is never even tried. This for instance allows users to write down expressions like `i >= 0 && a[i] == 'X'` without causing errors. If we were to replace this expression by `tmp = (a[i] == 'X'), i >= 0 && tmp` it would potentially create an error that wasn't present in the original expression. To avoid this, we will add a syntax `x =` e $\langle\varphi\rangle$ where the assignment is only executed when φ evaluates to non-zero. For simplicity, we will write `x =` e for `x =` e $\langle 1\rangle$.

5.2 Sequential Expressions

To flesh out these ideas, we define sequential expressions as follows:

Definition 1. A *conditional assignment* has one of the following forms:

- if `x` is a variable of a base type (not necessarily declared) and `f(`e_1`, ..., `e_n`)` a valid function call of the same type with all e_i pure expressions, and if φ is a pure expression of type `int`, then `x = f(`e_1`, ..., `e_n`)` $\langle\varphi\rangle$ is a conditional assignment;
- if `x` is a (not necessarily declared) variable of a base type, e a pure expression of the same type and φ a pure expression of type `int`, then `x =` e $\langle\varphi\rangle$ is a conditional assignment;
- if `x` is a declared variable of an (array X) type, i a pure expression of type `int`, e a pure expression of type X and φ a pure expression of type `int`, then `x[`i`] =` e $\langle\varphi\rangle$ is a conditional assignment.

An *assignment sequence* is a (possibly empty) sequence $e_1, \ldots, e_n$ of conditional assignments.

A *sequential expression* is a pair a, e of an assignment sequence and a pure expression. We will often simply denote e for ϵ, e.

We also use the `,` to denote sequence appending, so for instance $(a, b), (c, d)$ is simply a, b, c, d.

5.3 Translating Expressions

Let us now turn to the translation in question. The idea is simply what we have stated before: we take all the assignments out of the main expression and move them to the assignment sequence; only a pure expression remains.

- $n \mapsto \epsilon, n$;
- $\texttt{x} \mapsto \epsilon, \texttt{x}$;
- $\texttt{x[}e\texttt{]} \mapsto a, \texttt{x[}e'\texttt{]}$ if $e \mapsto a, e'$;
- $\texttt{-}e \mapsto a, \texttt{-}e'$ and $\texttt{+}e \mapsto a, e'$ and $\texttt{!}e \mapsto a, \texttt{!}e'$ if $e \mapsto a, e'$;
- $e_1, e_2 \mapsto \texttt{(}a_1\texttt{,}a_2\texttt{)}, e_2'$ if $e_1 \mapsto a_1, e_1'$ and $e_2 \mapsto a_2, e_2'$;
- $e_1 \circ e_2 \mapsto \texttt{(}a_1\texttt{,}a_2\texttt{)}, e_1' \circ e_2'$ if $e_1 \mapsto a_1, e_1'$ and $e_2 \mapsto a_2, e_2'$ for $\circ$ generated by Op;
- $e_1 \circ\ e_2 \mapsto \texttt{(}a_1\texttt{,}a_2'\texttt{)}, e_1' \circ e_2'$ if $e_1 \mapsto a_1, e_1'$ and $e_2 \mapsto a_2, e_2'$ for $\circ \in \{\texttt{\&\&}, \texttt{||}\}$ where a_2' is a_2 with all conditions φ of the conditional assignments replaced as follows: if $\varphi = 1$ then it is replaced by e_1', otherwise by φ `&&` e_1';
- $\texttt{f(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mapsto \texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{,x = f(}e_1'\texttt{,}\ldots\texttt{,}e_n'\texttt{)}\ \langle 1\rangle\texttt{),x}$ with x fresh, if each $e_i \mapsto a_i, e_i'$;
- $\texttt{(}e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3\texttt{)} \mapsto \texttt{(}a_1\texttt{,}a_2'\texttt{,}a_3'\texttt{),(}e_1'\ \texttt{?}\ e_2'\ \texttt{:}\ e_3'\texttt{)}$ if each $e_i \mapsto a_i, e_i'$ where a_2' is a_2 with all conditions φ of the conditional assignments replaced as follows: if $\varphi = 1$ then it is replaced by e_1', otherwise by φ `&&` e_1' and a_3' is a_3 with all conditions φ of the conditional assignments replaced as follows: if $\varphi = 1$ then it is replaced by $\texttt{!}e_1'$, otherwise by φ `&&` $\texttt{!}e_1'$;
- $\texttt{x =}\ e \mapsto \texttt{(}a\texttt{,x =}\ e'\ \langle 1\rangle\texttt{),x}$ if $e \mapsto a, e'$;
- $\texttt{x[}i\texttt{] =}\ e \mapsto \texttt{(}a\texttt{,}b\texttt{,x[}i'\texttt{] =}\ e'\ \langle 1\rangle\texttt{),x[}i'\texttt{]}$ if $i \mapsto a, i'$ and $e \mapsto b, e'$.

Example 1. The expression `(x = 3) + a[y]` is translated as follows:

- $\texttt{3} \mapsto \epsilon, \texttt{3}$
- $\texttt{x = 3} \mapsto \texttt{x = 3}\ \langle 1\rangle\texttt{,x}$
- $\texttt{y} \mapsto \epsilon, \texttt{y}$
- $\texttt{a[y]} \mapsto \epsilon, \texttt{a[y]}$
- $\texttt{(x = 3) + a[y]} \mapsto \texttt{x = 3}\ \langle 1\rangle\texttt{, x + a[y]}$

5.4 Usage

We will not use this transformation to sequential expressions as a preparsing step, but rather we define $\theta(e)$ for any expression e as the sequential expression such that $e \mapsto \theta(e)$.

6 Parsing Pure Expressions

In C, there is no separate boolean type; rather, anything non-zero is considered "true" for the purposes of boolean evaluation. In LCTRSs, however, we do have this distinction. To handle this discrepancy, we define two functions, interpreting a pure C-expression either as a boolean or an integer.

$$
\begin{aligned}
[\![n]\!]_{\texttt{Bool}} &= \texttt{false} \text{ if } n = 0 \text{ and } \texttt{true} \text{ otherwise} \\
[\![\texttt{x}]\!]_{\texttt{Bool}} &= \texttt{x} \neq \texttt{0} \\
[\![\texttt{x}[e]]\!]_{\texttt{Bool}} &= \texttt{select}(\texttt{x}, [\![e]\!]_{\texttt{Int}}) \neq \texttt{0} \\
[\![\texttt{-}e]\!]_{\texttt{Bool}} &= [\![e]\!]_{\texttt{Bool}} \\
[\![\texttt{+}e]\!]_{\texttt{Bool}} &= [\![e]\!]_{\texttt{Bool}} \\
[\![e_1 \circ e_2]\!]_{\texttt{Bool}} &= [\![e_1]\!]_{\texttt{Int}} \mathbin{\overline{\circ}} [\![e_2]\!]_{\texttt{Int}} \neq \texttt{0} \text{ for } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \\
[\![e_1 \circ e_2]\!]_{\texttt{Bool}} &= [\![e_1]\!]_{\texttt{Int}} \mathbin{\overline{\circ}} [\![e_2]\!]_{\texttt{Int}} \text{ for } \circ \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\
[\![\texttt{!}e]\!]_{\texttt{Bool}} &= \texttt{not}([\![e]\!]_{\texttt{Bool}}) \\
[\![e_1 \circ e_2]\!]_{\texttt{Bool}} &= [\![e_1]\!]_{\texttt{Bool}} \mathbin{\overline{\circ}} [\![e_2]\!]_{\texttt{Bool}} \text{ for } \circ \in \{\texttt{\&\&}, \texttt{||}\} \\
[\![\texttt{(}e_1 \texttt{ ? } e_2 \texttt{ : } e_3\texttt{)}]\!]_{\texttt{Bool}} &= ([\![e_1]\!]_{\texttt{Bool}} \wedge [\![e_2]\!]_{\texttt{Bool}}) \vee (\texttt{not}([\![e_1]\!]_{\texttt{Bool}}) \wedge [\![e_3]\!]_{\texttt{Bool}})
\end{aligned}
$$

$$
\begin{aligned}
[\![n]\!]_{\texttt{Int}} &= n \\
[\![\texttt{x}]\!]_{\texttt{Int}} &= \texttt{x} \\
[\![\texttt{x}[e]]\!]_{\texttt{Int}} &= \texttt{select}(\texttt{x}, [\![e]\!]_{\texttt{Int}}) \\
[\![\texttt{-}e]\!]_{\texttt{Int}} &= \texttt{0} - [\![e]\!]_{\texttt{Int}} \\
[\![\texttt{+}e]\!]_{\texttt{Int}} &= [\![e]\!]_{\texttt{Int}} \\
[\![e_1 \circ e_2]\!]_{\texttt{Int}} &= [\![e_1]\!]_{\texttt{Int}} \mathbin{\overline{\circ}} [\![e_2]\!]_{\texttt{Int}} \text{ for } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \\
[\![e_1 \circ e_2]\!]_{\texttt{Int}} &= \texttt{ite}([\![e_1]\!]_{\texttt{Int}} \mathbin{\overline{\circ}} [\![e_2]\!]_{\texttt{Int}}, \texttt{1}, \texttt{0}) \text{ for } \circ \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\
[\![\texttt{!}e]\!]_{\texttt{Int}} &= \texttt{ite}([\![e]\!]_{\texttt{Bool}}, \texttt{0}, \texttt{1}) \\
[\![e_1 \circ e_2]\!]_{\texttt{Int}} &= \texttt{ite}([\![e_1]\!]_{\texttt{Bool}} \mathbin{\overline{\circ}} [\![e_2]\!]_{\texttt{Bool}}, \texttt{1}, \texttt{0}) \text{ for } \circ \in \{\texttt{\&\&}, \texttt{||}\} \\
[\![\texttt{(}e_1 \texttt{ ? } e_2 \texttt{ : } e_3\texttt{)}]\!]_{\texttt{Int}} &= \texttt{ite}([\![e_1]\!]_{\texttt{Bool}}, [\![e_2]\!]_{\texttt{Int}}, [\![e_3]\!]_{\texttt{Int}})
\end{aligned}
$$

Here, $\overline{\circ}$ is the corresponding symbol in the local signature we use, e.g. $\overline{/}$ is `div`.

Furthermore, we will need to test whether any errors occur in an expression, in particular division-by-zero or array-out-of-bounds problems. To this end, we

define the function errorfree for pure expressions:

$$\begin{aligned}
\mathsf{errorfree}(n) &= \texttt{true} \\
\mathsf{errorfree}(\texttt{x}) &= \texttt{true} \\
\mathsf{errorfree}(\texttt{x}[e]) &= \mathsf{errorfree}(e) \wedge [\![e]\!]_{\texttt{Int}} \geq 0 \wedge [\![e]\!]_{\texttt{Int}} < \texttt{size}(\texttt{x}) \\
\mathsf{errorfree}(\texttt{-}e) &= \mathsf{errorfree}(e) \\
\mathsf{errorfree}(\texttt{+}e) &= \mathsf{errorfree}(e) \\
\mathsf{errorfree}(\texttt{!}e) &= \mathsf{errorfree}(e) \\
\mathsf{errorfree}(e_1 \circ e_2) &= \mathsf{errorfree}(e_1) \wedge \mathsf{errorfree}(e_2) \text{ for } \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\
\mathsf{errorfree}(e_1 \circ e_2) &= \mathsf{errorfree}(e_1) \wedge \mathsf{errorfree}(e_2) \wedge [\![e_2]\!]_{\texttt{Int}} \neq 0 \text{ for } \circ \in \{\texttt{/}, \texttt{\%}\} \\
\mathsf{errorfree}(e_1 \texttt{ \&\& } e_2) &= \mathsf{errorfree}(e_1) \wedge (\texttt{not}([\![e_1]\!]_{\texttt{Bool}}) \vee \mathsf{errorfree}(e_2)) \\
\mathsf{errorfree}(e_1 \texttt{ || } e_2) &= \mathsf{errorfree}(e_1) \wedge ([\![e_1]\!]_{\texttt{Bool}} \vee \mathsf{errorfree}(e_2)) \\
\mathsf{errorfree}(\texttt{(}e_1 \texttt{ ? } e_2 \texttt{ : } e_3\texttt{)}) &= \mathsf{errorfree}(e_1) \wedge (\texttt{not}([\![e_1]\!]_{\texttt{Bool}}) \vee \mathsf{errorfree}(e_2)) \wedge ([\![e_1]\!]_{\texttt{Bool}} \vee \mathsf{errorfree}(e_3))
\end{aligned}$$

7 Input and Output from Functions

To handle the use and updates of global variables in functions, the LCTRS we will create passes global and altered variables along in input and output variables to functions.

Now, for every function declaration $ResT\ \texttt{f}(C_1\ \texttt{x}_1, \ldots, C_n\ \texttt{x}_n)\ \{\ Ss\ \}$ in the program, let us denote $\mathsf{reads}(\texttt{f}) = \{\texttt{y}_1, \ldots, \texttt{y}_m\}$, ordered by occurrence in the program; that is, $\mathsf{globals}(\texttt{y}_i) < \mathsf{globals}(\texttt{y}_{i+1})$ for all $i < m$. Furthermore, let $\mathsf{writes}(\texttt{f}) = \{\texttt{y}_{i_1}, \ldots, \texttt{y}_{i_k}\}$, with each $i_j < i_{j+1}$ and $\mathsf{alters}(\texttt{f}) = \{j_1, \ldots, j_p\}$ with each $j_i < j_{i+1}$.

Function Declarations For any additional sequence $\texttt{z}_1, \ldots, \texttt{z}_l$ of locally declared variables, each declared with a sort $\kappa_1, \ldots, \kappa_l$, we let:

- $\mathsf{VarSequence}(\texttt{f}, \vec{\texttt{z}})$ is $[\texttt{y}_1, \ldots, \texttt{y}_m, \texttt{x}_1, \ldots, \texttt{x}_n, \texttt{z}_1, \ldots, \texttt{z}_l]$; this indicates the order in which we will use variables in symbols corresponding to positions in this function.
- $\mathsf{ReturnSequence}(\texttt{f})$ is $[\texttt{y}_{i_1}, \ldots, \texttt{y}_{i_k}, \texttt{x}_{j_1}, \ldots, \texttt{x}_{j_p}]$; this indicates the order of return variables (note that locally declared variables are not included, as these can only be returned as the *explicit* return value);
- $\mathsf{ReturnTypeDec}(\texttt{f})$ is $[sort(\texttt{y}_{i_1}) \times \cdots \times sort(\texttt{y}_{i_k}) \times sort(\texttt{x}_{j_1}) \times \cdots \times sort(\texttt{x}_{j_p})] \Rightarrow \texttt{result}_\texttt{f}$ if $ResT = \texttt{void}$ and $[\texttt{Int} \times sort(\texttt{y}_{i_1}) \times \cdots \times sort(\texttt{y}_{i_k}) \times sort(\texttt{x}_{j_1}) \times \cdots \times sort(\texttt{x}_{j_p})] \Rightarrow \texttt{result}_\texttt{f}$ if $ResT = \texttt{int}$.

Additionally, writing $\vec{\texttt{u}}$ for any sequence of variables, possibly including the global and parameter variables:

- $\mathsf{TypeDec}(\texttt{f}, \vec{\texttt{u}})$ is $[sort(\texttt{u}_1) \times \cdots \times sort(\texttt{u}_{|\vec{\texttt{u}}|})] \Rightarrow \texttt{result}_\texttt{f}$; this indicates the type declaration which we will use for symbols corresponding to positions in this function.

Function Calls For function calls $\mathtt{f}(e_1, \ldots, e_n)$ inside a (possibly different) function $\mathtt{g}$, at a position where the local variables (both arguments and locally declared variables) are $\mathtt{w}_1, \ldots, \mathtt{w}_q$ and $\mathsf{reads}(\mathtt{g}) = \{\mathtt{u}_1, \ldots, \mathtt{u}_h\}$ (ordered by occurrence in the program, so each $\mathsf{globals}(\mathtt{u}_i) < \mathsf{globals}(\mathtt{u}_{i+1})$; note that since $\mathsf{reads}(\mathtt{f}) \subseteq \mathsf{reads}(\mathtt{g})$ there is some i for every $j \in \{1, \ldots, m\}$ such that $\mathtt{u}_i = \mathtt{y}_j$), where $e_1, \ldots, e_n$ are pure expressions, we let:

- $\mathsf{PassedTerms}(\mathtt{f}, \vec{e}) = [\mathtt{y}_1, \ldots, \mathtt{y}_m, [\![e_1]\!]_{\mathtt{Int}}, \ldots, [\![e_n]\!]_{\mathtt{Int}}]$; these are the parameters which will be passed as input to the main function symbol $\mathtt{f}_0$;
- $\mathsf{ReturnedVars}(\mathtt{f}, \vec{e}) = [\mathtt{y}_{i_1}, \ldots, \mathtt{y}_{i_k}, e_{j_1}, \ldots, e_{j_p}]$ – this is a sequence of distinct variables, because (after pre-parsing) only distinct non-global array variables can be passed as array arguments of a function; these are the variables returned by the function call, following their names inside $\mathtt{g}$;
- $\mathsf{UntouchedVars}(\mathtt{f}, \vec{e}, \vec{\mathtt{u}\mathtt{w}}) = ([\mathtt{u}_1, \ldots, \mathtt{u}_h, \mathtt{w}_1, \ldots, \mathtt{w}_q] - [\mathtt{y}_{i_1}, \ldots, \mathtt{y}_{i_k}, e_{j_1}, \ldots, e_{j_p}])$, that is, the sequence $\vec{\mathtt{u}\mathtt{w}}$ modified by removing the variables in $\mathsf{writes}(\mathtt{f})$ and those altered by $\mathtt{f}$, but leaving the order the same as before;
- $\mathsf{UntouchedSorts}(\mathtt{f}, \vec{e}, \vec{\mathtt{u}\mathtt{w}}) = [sort(\mathtt{v}_1), \ldots, sort(\mathtt{v}_{|\vec{\mathtt{v}}|})]$ where $\mathsf{UntouchedVars}(\mathtt{f}, \vec{e}, \vec{\mathtt{u}\mathtt{w}}) = \vec{\mathtt{v}}$.

These definitions are given here because we will need them multiple times to convert various kinds of statements in the program, as we will do in the next section.

8 Converting the Program

In this section, we define a conversion from preparsed C-programs to an LCTRS.

8.1 Sorts and Primary Function Symbols

To start, we define the "main" signature: the signature of function symbols reflecting all the functions. Recall that the program (after preparsing) has the form $D_1 \ldots D_k \; F_1 \ldots F_m$ where the D_i are global variable declarations (which we already stored into the $\mathsf{globals}$ mapping and defined in $\mathcal{V}$), and the F_i are complete functions, each with a distinct name (unlike C++, C does not permit multiple functions having the same name[1]).

Now, for every function F_i of the form $ResT \;\; \mathtt{f(}C_1 \;\; \mathtt{x}_1\mathtt{,} \ldots\mathtt{,} \;\; C_n \;\; \mathtt{x}_n\mathtt{)} \;\; \mathtt{\{\ldots\}}$, we will create the following function symbols for the signature:

- $\mathtt{f}_0 : \mathsf{TypeDec}(\mathtt{f}, \mathsf{VarSequence}(\mathtt{f}, []))$ (additional symbols $\mathtt{f}_i$ will be introduced later);
- $\mathsf{ret}_{\mathtt{f}} : \mathsf{ReturnTypeDec}(\mathtt{f})$;
- $\mathsf{err}_{\mathtt{f}} : \mathtt{result}_{\mathtt{f}}$.

[1] although it would not be too problematic if this was allowed: we would simply have renamed such functions during the preparsing step

8.2 Converting Individual Functions

The body of a function – a list of statements – is converted using the function convert. This function also handles partial functions and *blocks* of code, and takes seven arguments to do so:

- `f`, the name of the function;
- i, the number of symbols already declared for this function;
- $\vec{x} = [x_1, \ldots, x_n]$, a list of variables (in $\mathcal{V}$) which are known at the start of the function or block;
- $\vec{y} = [y_1, \ldots, y_m]$, a list of variables which have been locally declared (and should therefore be forgotten at the end of the function or block);
- Σ: the function symbols declared so far;
- $\mathcal{R}$: the rules declared so far;
- Ss, the statements to convert.

The function returns a tuple:

- Σ': the function symbols declared for these statements, added to Σ;
- $\mathcal{R}'$: the rules defined for these statements, added to $\mathcal{R}$;
- j, the last index of symbols declared for these statements.

The set Σ' will often contain helping symbols `continue` and `break`, which should be replaced by the corresponding symbols after parsing a loop block.

Assuming this convert function defined, the translation of a function definition $ResT$ `f`$(C_1$ `x`$_1, \ldots, C_n$ `x`$_n)\{Ss\}$ is obtained as follows:

- let $\mathsf{convert}(\mathtt{f}, 0, \mathsf{VarSequence}(\mathtt{f}, []), [], \emptyset, \emptyset, Ss) = \langle \Sigma, \mathcal{R}, j \rangle$; then Σ cannot contain `continue` or `break` as neither statement is allowed to occur outside a loop;
- the set of function symbols generated by this function, Σ', is obtained from Σ by removing $\mathtt{f}_j$, if it is included;
- the set of rules generated by this function, $\mathcal{R}'$, is obtained from $\mathcal{R}$ by replacing all occurrences of $\mathtt{f}_j(\vec{y})$ in $\mathcal{R}$ by
  - $\mathsf{ret}_\mathtt{f}(\vec{z})$ where $[\vec{z}] = \mathsf{ReturnSequence}(\mathtt{f})$ if $ResT =$ `void`;
  - $\mathsf{ret}_\mathtt{f}(\mathtt{rnd}, \vec{z})$ where `rnd` is a fresh variable of sort `Int` and $[\vec{z}] = \mathsf{ReturnSequence}(\mathtt{f})$ if $ResT =$ `int`.

We convert the program by thus converting all functions, and taking the unions of the resulting sets of signatures and rules, along with the already-defined initial symbols $\mathtt{f}_0$, $\mathsf{ret}_\mathtt{f}$, and $\mathsf{err}_\mathtt{f}$. For the `main` function, we moreover start by initializing all variables to

We will now consider how to handle all kinds of statements. To start, we define $\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \epsilon) = (\Sigma, \mathcal{R}, i)$; that is, the alphabet, rules and current function symbol index are passed unmodified. For non-empty sequences of statements, we consider the first statement in particular.

Expressions To convert expressions, we first declare a helping function, which converts an assignment sequence.

- $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i, \vec{\mathtt{z}}, \vec{\mathtt{w}}, \Sigma, \mathcal{R}, \epsilon) = \langle i, \vec{\mathtt{w}}, \Sigma, \mathcal{R} \rangle$;
- $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i, \vec{\mathtt{z}}, \vec{\mathtt{w}}, \Sigma, \mathcal{R}, s \texttt{ = } e\ \langle \varphi \rangle \texttt{,} \vec{a}) =$
  - If e is a pure expression, then there are only two possibilities: s is some $\mathtt{z}_j$ (that is, a declared variable, not one which was only introduced in the assignment sequence), or s is $\mathtt{z}_j[k]$ for some declared variable and pure expression k.

    In the first case, $s = \mathtt{z}_j$, we let $\vec{\mathtt{u}}$ be $\vec{\mathtt{z}}$ with $\mathtt{z}_j$ replaced by $[\![e]\!]_{\mathtt{Int}}$, and let $\psi := \mathsf{errorfree}(e)$.

    In the second case, we let $\vec{\mathtt{u}}$ be $\vec{\mathtt{z}}$ with $\mathtt{z}_j$ replaced by $\mathtt{store}(\mathtt{z}_j, [\![k]\!]_{\mathtt{Int}}, [\![e]\!]_{\mathtt{Int}})$, and let $\psi := \mathsf{errorfree}(\mathtt{z}[k]) \wedge \mathsf{errorfree}(e)$.

    In either case, the result is $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i+1, \vec{\mathtt{z}}, \vec{\mathtt{w}}, \Sigma', \mathcal{R}', \vec{a})$, where
    - $\Sigma' = \Sigma \cup \{\mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, \vec{\mathtt{z}}\vec{\mathtt{w}})\}$
    - $\mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{z}}, \vec{\mathtt{w}}) \to \mathtt{f}_{i+1}(\vec{\mathtt{u}}, \vec{\mathtt{w}})\ [[\![\varphi]\!]_{\mathtt{Bool}} \wedge \psi],$
      $\mathtt{f}_i(\vec{\mathtt{z}}, \vec{\mathtt{w}}) \to \mathtt{f}_{i+1}(\vec{\mathtt{z}}, \vec{\mathtt{w}})\ [\mathtt{not}([\![\varphi]\!]_{\mathtt{Bool}})],$
      $\mathtt{f}_i(\vec{\mathtt{z}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [[\![\varphi]\!]_{\mathtt{Bool}} \wedge \mathtt{not}(\psi)]\}$

    Note that we do not error-check the constraint φ here, since it will always occur as a subterm of a pure expression later on, so will be checked regardless.
  - Otherwise, e must be a function call $\mathtt{g}(e_1, \ldots, e_n)$ and s a fresh variable. Write $\psi = \mathsf{errorfree}(e_1) \wedge \ldots \wedge \mathsf{errorfree}(e_n)$. Also, let:
    - $\mathsf{PassedTerms}(\mathsf{g}, \vec{e}) = [p_1, \ldots, p_k]$, so the sequence of global variables followed by interpreted expressions to be passed to g;
    - $\mathsf{ReturnedVars}(\mathsf{g}, \vec{e}) = [\mathtt{v}_1, \ldots, \mathtt{v}_p]$, so the global variables and array arguments which may have been updated by the call;
    - $\mathsf{UntouchedVars}(\mathsf{g}, \vec{e}, \vec{\mathtt{z}}\vec{\mathtt{w}}) = [\mathtt{u}_1, \ldots, \mathtt{u}_q]$, so those variables in $\vec{\mathtt{z}}\vec{\mathtt{w}}$, left in the original order, which do not occur in $\mathsf{ReturnedVars}(\mathsf{g}, \vec{e})$;
    - $\mathsf{UntouchedSorts}(\mathsf{g}, \vec{e}, \vec{\mathtt{z}}\vec{\mathtt{w}}) = [t_1, \ldots, t_q]$.

    In this case, we assume the fresh variable s to be declared in $\mathcal{V}$ with sort $\mathtt{Int}$. The result is $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i+2, \vec{\mathtt{z}}, [\vec{\mathtt{w}}, s], \Sigma', \mathcal{R}', \vec{a})$, where:
    - $\Sigma' = \Sigma \cup \{\mathtt{f}_{i+1} : [t_1 \times \ldots \times t_q \times \mathtt{result}_{\mathsf{g}}] \Rightarrow \mathtt{result}_{\mathtt{f}},$
      $\mathtt{f}_{i+2} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{z}}, \vec{\mathtt{w}}, s])\}$
    - $\mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{z}}, \vec{\mathtt{w}}) \to \mathtt{f}_{i+1}(\mathtt{u}_1, \ldots, \mathtt{u}_q, \mathsf{g}_0(p_1, \ldots, p_k))\ [[\![\varphi]\!]_{\mathtt{Bool}} \wedge \psi],$
      $\mathtt{f}_{i+1}(\mathtt{u}_1, \ldots, \mathtt{u}_q, \mathsf{ret}_{\mathsf{g}}(\mathtt{x}, \mathtt{v}_1, \ldots, \mathtt{v}_p)) \to \mathtt{f}_{i+2}(\vec{\mathtt{z}}, \vec{\mathtt{w}}, \mathtt{x}),$
      $\mathtt{f}_{i+1}(\mathtt{u}_1, \ldots, \mathtt{u}_q, \mathsf{err}_{\mathsf{g}}) \to \mathsf{err}_{\mathtt{f}}$
      $\mathtt{f}_i(\vec{\mathtt{z}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [[\![\varphi]\!]_{\mathtt{Bool}} \wedge \mathtt{not}(\psi)]$
      $\mathtt{f}_i(\vec{\mathtt{z}}, \vec{\mathtt{w}}) \to \mathtt{f}_{i+2}(\vec{\mathtt{z}}, \vec{\mathtt{w}}, \mathtt{rnd})\ [\mathtt{not}([\![\varphi]\!]_{\mathtt{Bool}})]\}$

    where $\mathtt{rnd}$ is a fresh variable of sort $\mathtt{Int}$ (representing a random choice). Note that the second rule does not introduce fresh variables, since $\{\vec{\mathtt{u}}, \vec{\mathtt{v}}\}$ is exactly $\{\vec{\mathtt{z}}, \vec{\mathtt{w}}\}$.

To convert a statement *exp*;, we simply calculate $\theta(exp)$ and execute its assignment sequence using $\mathsf{convert}_{\mathtt{exp}}$, then drop the extra variables. Formally:

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, e; Ss) &:= \mathsf{convert}(\mathtt{f}, j, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma'', \mathcal{R}'', Ss)\\ \text{if } \theta(e) = \vec{a}\,\mathtt{,}\,e' \text{ and } \mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, \vec{a}) &= \langle j, \vec{\mathtt{w}}, \Sigma', \mathcal{R}'\rangle\\ \text{and } \Sigma'' = (\Sigma' \setminus \{\mathtt{f}_j\}) &\cup \{\mathtt{f}_j : \mathsf{TypeDec}(f, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\}\\ \text{and } \mathcal{R}'' = \mathcal{R}' \text{ with occurrences of } \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) &\text{ replaced by } \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}})\end{aligned}$$

The replacement is safe because the last symbol $\mathtt{f}_j$ can only ever occur on the right-hand side of a rule in $\mathsf{convert}_{\mathtt{exp}}$.

Declarations In the preparsing step, we have altered all declarations such that only one variable is declared at a time. Thus, declarations must have one of the following forms:

- `int x`
- `int x`$[n]$
- `int x` $= e$
- `int x`$[n] = \{e_1, \ldots, e_k\}$ with $k \leq n$
- `int x`$[] = \{e_1, \ldots, e_k\}$

We convert such statements by adding a new position for a local variable, and – if applicable – initializing it with the interpretation of the expression(s).

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{int\ z;}\, Ss) &:= \mathsf{convert}(\mathtt{f}, i+1, \vec{\mathtt{x}}, [\vec{\mathtt{y}}, \mathtt{z}], \Sigma', \mathcal{R}', Ss)\\ &\text{where } \Sigma' = \Sigma \cup \{\mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z}])\},\\ &\text{and } \mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{f}_{i+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{rnd})\}\end{aligned}$$

Here, again, `rnd` is a variable of type `Int` that does not occur in $[\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z}]$. Similarly, letting `rndarr` be a variable of type `IntArray` that isn't used in the current variables:

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{int\ z}[n]\mathtt{;}\, Ss) &:= \mathsf{convert}(\mathtt{f}, i+1, \vec{\mathtt{x}}, [\vec{\mathtt{y}}, \mathtt{z}], \Sigma', \mathcal{R}', Ss)\\ &\text{where } \Sigma' = \Sigma \cup \{\mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z}])\},\\ &\text{and } \mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{f}_{i+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{rndarr})\ [\mathtt{size}(\mathtt{rndarr}) = n]\}\end{aligned}$$

If $\theta(e) = \vec{a}\,\mathtt{,}\,e'$ and $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, \vec{a}) = \langle j, \vec{\mathtt{w}}, \Sigma', \mathcal{R}'\rangle$, then:

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{int\ z} = e\mathtt{;}\, Ss) &:= \mathsf{convert}(\mathtt{f}, j+1, \vec{\mathtt{x}}, [\vec{\mathtt{y}}, \mathtt{z}], \Sigma'', \mathcal{R}'', Ss)\\ &\text{where } \Sigma'' = \Sigma' \cup \{\mathtt{f}_{j+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z}])\},\\ &\text{and } \mathcal{R}'' = \mathcal{R}' \cup \{\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{i+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}}, [\![e]\!]_{\mathtt{Int}})\ [\mathsf{errorfree}(e)],\\ &\qquad \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\mathsf{errorfree}(e))]\}\end{aligned}$$

For array initialisations, we consider a special case when encountering a trivial initialisation: $\mathtt{x}[k] = \{n_1, \ldots, n_k\}$ where all n_i are integers (also simply presented as $\mathtt{x}[] = \{n_1, \ldots, n_k\}$). In this case, the given array is a *value*, and can be initialised as such in the LCTRS:

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{int\ z}[k] = \{n_1, \ldots, n_k\}\mathtt{;}\, Ss) &:=\\ &\mathsf{convert}(\mathtt{f}, i+1, \vec{\mathtt{x}}, [\vec{\mathtt{y}}, \mathtt{z}], \Sigma', \mathcal{R}', Ss)\\ &\text{where } \Sigma' = \Sigma \cup \{\mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z}])\},\\ &\text{and } \mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{f}_{i+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \{n_1, \ldots, n_k\})\}\end{aligned}$$

For the general case, $\mathtt{x}[n] = \{e_1, \ldots, e_k\}$ (or $\mathtt{x}[] = \{e_1, \ldots, e_k\}$ where the e_i are not all values – we will consider this case as the former one, where $n = k$), we do not automatically have a corresponding value. Thus, the initialisation is done in the constraint. Let $\theta(e_i) = \vec{a_i}, e'_i$ for all $1 \leq i \leq k$, and assume the fresh variables in each a_i are chosen disjointly. Then, if $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, [\vec{a_1}, \ldots, \vec{a_k}]) = \langle j, \vec{\mathtt{w}}, \Sigma', \mathcal{R}' \rangle$ and ψ denotes the formula $\mathsf{errorfree}(e'_1) \wedge \cdots \wedge \mathsf{errorfree}(e'_k)$, we have:

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{int\ z}[n] = \{e_1, \ldots, e_k\}\mathtt{;}\, Ss) := \\
&\qquad \mathsf{convert}(\mathtt{f}, j+1, \vec{\mathtt{x}}, [\vec{\mathtt{y}}, \mathtt{z}], \Sigma'', \mathcal{R}'', Ss) \\
&\qquad \text{where } \Sigma'' = \Sigma' \cup \{\mathtt{f}_{j+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z}])\}, \\
&\qquad \text{and } \mathcal{R}'' = \mathcal{R}' \cup \{\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{j+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{z})\ [\psi \wedge \mathtt{size(z)} = n \wedge \\
&\qquad\qquad \mathtt{select(z, 0)} = e'_1 \wedge \cdots \wedge \mathtt{select(z, k\text{-}1)} = e'_k], \\
&\qquad\qquad \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\psi)]\}
\end{aligned}$$

Function Calls Function calls as statements (for `Void` functions) are treated very similarly to integer function calls inside expressions. For $1 \leq i \leq n$, let $\theta(e_i) = \vec{a_i}, e'_i$. Additionally, let $\mathsf{convert}_{\mathtt{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, [\vec{a_1}, \ldots, \vec{a_k}]) = \langle j, \vec{\mathtt{w}}, \Sigma', \mathcal{R}' \rangle$, let ψ denote the formula $\mathsf{errorfree}(e_1) \wedge \cdots \wedge \mathsf{errorfree}(e_k)$, and let:

- $\mathsf{PassedTerms}(\mathtt{g}, \vec{e}) = [p_1, \ldots, p_k]$ (expressions to be passed to $\mathtt{g}$);
- $\mathsf{ReturnedVars}(\mathtt{g}, \vec{e}) = [\mathtt{v}_1, \ldots, \mathtt{v}_p]$ (variables returned by $\mathtt{g}$);
- $\mathsf{UntouchedVars}(\mathtt{g}, \vec{e}, \vec{\mathtt{x}}\vec{\mathtt{y}}) = [\mathtt{u}_1, \ldots, \mathtt{u}_q]$ (variables not affected by the call);
- $\mathsf{UntouchedSorts}(\mathtt{g}, \vec{e}, \vec{\mathtt{x}}\vec{\mathtt{y}}) = [t_1, \ldots, t_q]$ (sorts of $\vec{u}$).

Having this, we define $\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, f\,\mathtt{(}e_1, \ldots, e_n\mathtt{)}\,\mathtt{;}\, Ss) := \mathsf{convert}(\mathtt{f}, j+2, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma'', \mathcal{R}'', Ss)$, where:

- $\Sigma'' = \Sigma' \cup \{\mathtt{f}_{j+1} : [t_1 \times \ldots \times t_q \times \mathtt{result}_{\mathtt{g}}] \Rightarrow \mathtt{result}_{\mathtt{f}},$
  $\mathtt{f}_{j+2} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\}$
- $\mathcal{R}'' = \mathcal{R}' \cup \{\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{j+1}(\mathtt{u}_1, \ldots, \mathtt{u}_q, \mathtt{g}_0(p_1, \ldots, p_k))\ [\psi],$
  $\mathtt{f}_{j+1}(\mathtt{u}_1, \ldots, \mathtt{u}_q, \mathsf{ret}_{\mathtt{g}}(\mathtt{v}_1, \ldots, \mathtt{v}_p)) \to \mathtt{f}_{j+2}(\vec{\mathtt{x}}, \vec{\mathtt{y}}),$
  $\mathtt{f}_{j+1}(\mathtt{u}_1, \ldots, \mathtt{u}_q, \mathsf{err}_{\mathtt{g}}) \to \mathsf{err}_{\mathtt{f}}$
  $\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\psi)]$

  Note that the second rule does not introduce fresh variables, since $\{\vec{\mathtt{u}}, \vec{\mathtt{v}}\}$ is exactly $\{\vec{\mathtt{x}}, \vec{\mathtt{y}}\} \cup \{\mathtt{w}_j \mid \mathtt{w}_j \text{ occurs in } \vec{\mathtt{p}}\}$.

Continue and Break While only allowed to occur inside loops, we assign an independent interpretation to `break;` and `continue;` which will then be replaced in the treatment of the various loop statements.

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{break;}\ Ss) := \langle \Sigma', \mathcal{R}', i+1 \rangle \\
&\text{where } \Sigma' = \Sigma \cup \{\mathtt{break} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}]), \mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\
&\text{and } \mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{break}(\vec{\mathtt{x}})\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{break;}\ Ss) := \langle \Sigma', \mathcal{R}', i+1 \rangle \\
&\text{where } \Sigma' = \Sigma \cup \{\mathtt{continue} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}]), \mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\
&\text{and } \mathcal{R}' = \mathcal{R} \cup \{\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{continue}(\vec{\mathtt{x}})\}
\end{aligned}$$

Thus we see: `break` and `continue` take as arguments only the variables $\vec{\mathtt{x}}$ which are already defined at the start of the block, not those locally declared. This guarantees that later occurrences of either statement inside the same block end up having the same declaration.

Note the additional introduction of $\mathtt{f}_{i+1}$; this is because we will handle some other statements by changing the "final" symbol $\mathtt{f}_j$ declared for a statement. Since it would be undesirable to change $\mathtt{f}_i$ when it is part of the left-hand side of the rule, we simply include an unused symbol.

Return Statements The `return` statement is handled very similarly to `break` and `continue`, except of course in that it may occur outside loop blocks. First, consider `return` without arguments, which might occur both in `void` and `int` functions.

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{return;}\ Ss) &:= \langle \Sigma', \mathcal{R}', i+1\rangle \\ \text{where } \Sigma' = \Sigma \cup \{&\mathtt{return} : \mathsf{ReturnTypeDec}(\mathtt{f}), \mathtt{f}_{i+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\ \text{and } \mathcal{R}' = \mathcal{R} \cup \{&\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{return}(\mathsf{ReturnSequence}(\mathtt{f}))\} \text{ if } \mathtt{f} \text{ has } \mathtt{void} \text{ type} \\ \text{or } \mathcal{R} \cup \{&\mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}}) \to \mathtt{return}(\mathit{rnd}, \mathsf{ReturnSequence}(\mathtt{f}))\} \text{ if } \mathtt{f} \text{ has } \mathtt{int} \text{ type}\end{aligned}$$

Here, *rnd* is a fresh variable. When an expression is given to return, we of course have to evaluate the expression first. To this end, let $\theta(e) = \vec{a}, e'$.

$$\begin{aligned}\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{return}\ e\mathtt{;}\ Ss) &:= \langle \Sigma'', \mathcal{R}'', j+1\rangle \\ \text{where } \mathsf{convert}_{\mathsf{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, \vec{a}) &= \langle j, \vec{\mathtt{w}}, \Sigma', \mathcal{R}'\rangle \\ \text{and } \Sigma'' = \Sigma' \cup \{&\mathtt{return} : \mathsf{ReturnTypeDec}(\mathtt{f}), \mathtt{f}_{j+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\ \text{and } \mathcal{R}'' = \mathcal{R}' \cup \{&\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{return}(e', \mathsf{ReturnSequence}(\mathtt{f}))\}\end{aligned}$$

Thus, all occurrences of `return` inside a function automatically have the same type declaration, and return the expected variables (note that `return` e can only occur in a function with non-void type, as the compiler would not accept it otherwise). As before, we include an unused symbol $\mathtt{f}_{i+1}$ or $\mathtt{f}_{j+1}$ to represent continuation from this point.

Blocks Blocks are groups of statements, allowing the declaration of local variables which are out of scope afterwards. To start, let:

- $\mathsf{convert}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, ss) = \langle \Sigma', \mathcal{R}', j\rangle$;
- Σ'' be Σ' with:
  - the type declaration of $\mathtt{f}_j$ replaced by $\mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])$
  - the type declaration of `break` and `continue` (if either of these is in Σ') replaced by $\mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}])$
- $\mathcal{R}''$ be $\mathcal{R}'$ with:
  - occurrences of $\mathtt{f}_j(e_1, \ldots, e_m)$ replaced by $\mathtt{f}_j(e_1, \ldots, e_{|\vec{\mathtt{x}}|+|\vec{\mathtt{y}}|})$
  - occurrences of $\mathtt{break}(e_1, \ldots, e_m)$ replaced by $\mathtt{break}(e_1, \ldots, e_{|\vec{\mathtt{x}}|})$
  - occurrences of $\mathtt{continue}(e_1, \ldots, e_m)$ replaced by $\mathtt{continue}(e_1, \ldots, e_{|\vec{\mathtt{x}}|})$

Then:

$$\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \{ss\}Ss) := \mathsf{convert}(\mathtt{f}, j, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma'', \mathcal{R}'', Ss)$$

That is, at the end of a block, all variables declared inside the block are removed from consideration. Thus, the symbol $\mathtt{f}_j$ at the end of a block necessarily has the same type declaration as the symbol $\mathtt{f}_i$ at its beginning. Moreover, the property that occurrences of `break` and `continue` and have a type declaration based on the *fixed* variables $\vec{\mathtt{x}}$ is preserved.

If-Statements To handle an `if` statement, we have to evaluate two distinct paths, as well as the expression. That is, to handle a statement `if(` e `) {` ss `} else {` tt `}` (all `if`-statements have such a form due to the preparsing), let:

- $\theta(e) = \vec{a}, e'$ and $\psi = \mathsf{errorfree}(e')$;
- $\mathsf{convert}_{\mathsf{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, \vec{a}) = \langle j, \vec{\mathtt{w}}, \Sigma_1, \mathcal{R}_1 \rangle$;
- $\mathsf{convert}(\mathtt{f}, j+1, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \emptyset, \emptyset, \{\ ss\ \}\epsilon) = \langle \Sigma_2, \mathcal{R}_2, k \rangle$;
- $\mathsf{convert}(\mathtt{f}, k, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \emptyset, \emptyset, \{\ tt\ \}\epsilon) = \langle \Sigma_3, \mathcal{R}_3, n \rangle$;
- Σ_2' be $\Sigma_2 \setminus \{\mathtt{f}_k\}$;
- $\mathcal{R}_2'$ be $\mathcal{R}_2$ with all occurrences of $\mathtt{f}_k$ replaced by $\mathtt{f}_n$.

Note that $\mathtt{f}_k$ has the same type declaration in Σ_2 as $\mathtt{f}_n$ in Σ_3: both are at the end of a block, so have the same type declaration as the beginning, which in both cases is $\mathsf{TypeDec}(\mathtt{f}, [\vec{x}, \vec{y}])$. We define:

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{if(}\ e\ \mathtt{)}\ \{\ ss\ \}\ \mathtt{else}\ \{\ tt\ \}\ Ss) := \\
&\qquad \mathsf{convert}(\mathtt{f}, n, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma', \mathcal{R}', Ss) \\
&\text{where } \Sigma' = \Sigma_1 \cup \Sigma_2' \cup \Sigma_3 \cup \{\mathtt{f}_{j+1}, \mathtt{f}_k : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\
&\text{and } \mathcal{R}' = \mathcal{R}_1 \cup \mathcal{R}_2' \cup \mathcal{R}_3 \cup \{\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{j+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge [\![e']\!]_{\mathtt{Bool}}], \\
&\qquad \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge \mathtt{not}([\![e']\!]_{\mathtt{Bool}})], \\
&\qquad \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\psi)]\}
\end{aligned}$$

Loops For each of `while`, `do` and `for`, we must evaluate the loop condition as well as the block. In each case, let $\theta(e) = \vec{a}, e'$ and $\psi = \mathsf{errorfree}(e')$. In addition, for starting index s and end index d which both have type declaration $\mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])$, let $\mathsf{Adapt}(\mathcal{R}, s, d)$ be $\mathcal{R}$ with occurrences of:

- $\mathtt{f}_d(\vec{e})$ replaced by $\mathtt{f}_s(\vec{e})$.
- $\mathtt{break}(\vec{e})$ replaced by $\mathtt{f}_d(\vec{e})$;
- $\mathtt{continue}(\vec{e})$ replaced by $\mathtt{f}_s(\vec{e})$;

Now, for `while`, we define:

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{while(}\ e\ \mathtt{)}\ \{\ ss\ \}\ Ss) := \mathsf{convert}(\mathtt{f}, k, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma''', \mathcal{R}''', Ss) \\
&\qquad \text{where } \mathsf{convert}_{\mathsf{exp}}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, \vec{a}) = \langle j, \vec{\mathtt{w}}, \Sigma', \mathcal{R}' \rangle \\
&\qquad \text{and } \mathsf{convert}(\mathtt{f}, j+1, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma', \mathcal{R}', \{\ ss\ \}\epsilon) = \langle k, \Sigma'', \mathcal{R}'' \rangle \\
&\qquad \text{and } \Sigma''' = (\Sigma'' \setminus \{\mathtt{break}, \mathtt{continue}\}) \cup \{\mathtt{f}_{j+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\
&\qquad \text{and } \mathcal{R}''' = \mathsf{Adapt}(\mathcal{R}'', i, k) \cup \{\mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{j+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge [\![e']\!]_{\mathtt{Bool}}], \\
&\qquad\qquad \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge \mathtt{not}([\![e']\!]_{\mathtt{Bool}})], \\
&\qquad\qquad \mathtt{f}_j(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\psi)]\}
\end{aligned}$$

Note the use of Adapt: any term reducing to the end of the block, $\mathtt{f}_k$, is diverted to the start, $\mathtt{f}_i$. In addition, breaks go to the end of the block, and continues to the beginning. The beginning is naturally the point before the condition is evaluated.

For `do`, we define:

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{do\{}\ ss\ \mathtt{\}\ while(}\ e\ \mathtt{);}\ Ss) := \\
&\qquad \mathsf{convert}(\mathtt{f}, k+1, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma''', \mathcal{R}''', Ss) \\
&\quad \text{where } \mathsf{convert}(\mathtt{f}, i, [\vec{x}, \vec{y}], [], \Sigma, \mathcal{R}, \{\ ss\ \}\epsilon) = \langle j, \Sigma', \mathcal{R}' \rangle \\
&\quad \text{and } \mathsf{convert}_{\mathsf{exp}}(\mathtt{f}, j, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma', \mathcal{R}', \vec{a}) = \langle k, \vec{\mathtt{w}}, \Sigma'', \mathcal{R}'' \rangle \\
&\quad \text{and } \Sigma''' = (\Sigma'' \setminus \{\mathtt{break}, \mathtt{continue}\}) \cup \{\mathtt{f}_{j+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\
&\quad \text{and } \mathcal{R}''' = \mathsf{Adapt}(\mathcal{R}'', i, k+1) \cup \{\mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_i(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge [\![e']\!]_{\mathtt{Bool}}], \\
&\qquad \mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{k+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge \mathtt{not}([\![e']\!]_{\mathtt{Bool}})], \\
&\qquad \mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\psi)]\}
\end{aligned}$$

And finally, for `for`, we define:

$$\begin{aligned}
&\mathsf{convert}(\mathtt{f}, i, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma, \mathcal{R}, \mathtt{for(}\ init\ \mathtt{;}\ e\ \mathtt{;}\ incr\ \mathtt{)}\ \{\ ss\ \}\ Ss) := \\
&\qquad \mathsf{convert}(\mathtt{f}, n, \vec{\mathtt{x}}, \vec{\mathtt{y}}, \Sigma_4, \mathcal{R}_4, Ss) \\
&\quad \text{where } \mathsf{convert}(\mathtt{f}, i, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma, \mathcal{R}, init\mathtt{;}\ \epsilon) = \langle j, \Sigma_1, \mathcal{R}_1 \rangle \\
&\quad \text{and } \mathsf{convert}_{\mathsf{exp}}(\mathtt{f}, j, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma_1, \mathcal{R}_1, \vec{a}) = \langle k, \vec{\mathtt{w}}, \Sigma_2, \mathcal{R}_2 \rangle \\
&\quad \text{and } \mathsf{convert}(\mathtt{f}, k+1, [\vec{\mathtt{x}}, \vec{\mathtt{y}}], [], \Sigma_2, \mathcal{R}_2, \{\ ss\ \}\ incr\mathtt{;}\epsilon) = \langle n, \Sigma_3, \mathcal{R}_3 \rangle \\
&\quad \text{and } \Sigma_4 = (\Sigma_3 \setminus \{\mathtt{break}, \mathtt{continue}\}) \cup \{\mathtt{f}_{k+1} : \mathsf{TypeDec}(\mathtt{f}, [\vec{\mathtt{x}}, \vec{\mathtt{y}}])\} \\
&\quad \text{and } \mathcal{R}_4 = \mathsf{Adapt}(\mathcal{R}_3, j, n) \cup \{\mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_{k+1}(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge [\![e']\!]_{\mathtt{Bool}}], \\
&\qquad \mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathtt{f}_n(\vec{\mathtt{x}}, \vec{\mathtt{y}})\ [\psi \wedge \mathtt{not}([\![e']\!]_{\mathtt{Bool}})], \\
&\qquad \mathtt{f}_k(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \vec{\mathtt{w}}) \to \mathsf{err}_{\mathtt{f}}\ [\mathtt{not}(\psi)]\}
\end{aligned}$$

References

[1] C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proceedings of the 9th International Symposium on Frontiers of Combining Systems*, volume 8152 of *LNAI*, pages 343–358, Nancy, 2013. Springer.