

Appendix D

C-with-Ease

D.1 *C-with-Ease* definition

At the highest level a *C-with-Ease* program consists of a collection of processes which interact via shared data structures.

The C language is enhanced by combining with the *Ease* process and process interaction model.

Ease provides simple and symmetric operators (read and write, get and put), a well defined process model. Constructions for both cooperative and subordinate concurrency and a mechanism (combinations) for building statically reusable and virtual resources on parallel and distributed machines.

The *Ease* keywords which appear in C programs begin with an *escape* character (generally %) to distinguish them from the names used in the standard C function library.

D.1.1 Process definition

A process differs from a C function since it does not return a value and obeys the *Ease* rules for parallel construction which place restrictions on the use of free variables and pointers.

A process is defined in a similar way to ANSI-C function definitions except where C functions specify the type of the returned value, processes are distinguished from functions by the keyword `%process`.

$$\left| \begin{array}{l}
 \text{definition} = \boxed{\text{process}} \text{ name } \boxed{(} \{ \boxed{0} \text{, } \boxed{\text{format}} \} \boxed{)} \\
 \qquad \qquad \qquad \text{body} \\
 \text{body} \qquad = \text{compound_statement} \\
 \text{process} \qquad = \text{name } \boxed{(} \boxed{\text{actual}} \boxed{)}
 \end{array} \right.$$

Since the body of a process does not return a value the body is simply terminated by the use of the C keyword `return`.

Names declared in a process definition may only appear in *Ease* process creation statements.

D.1.2 Parallel construction

$$\left| \begin{array}{l}
 \text{cooperation} = \boxed{\text{cooperate}} \boxed{\{ \langle \boxed{1} \text{; } \boxed{\text{process}} \rangle \}} \\
 \qquad \qquad \qquad | \boxed{\text{cooperate}} \boxed{(} \boxed{\text{replicator}} \boxed{)} \boxed{\text{process}} \boxed{;}
 \end{array} \right.$$

The processes in a cooperation start simultaneously and continue together. A cooperation terminates when all the components of the cooperation have terminated.

A free variable can be assigned to in only one component of a cooperation. If a free variable is assigned to, it can only appear in the assigning component. Otherwise, free variables can appear in the expressions of all components.

A context can be output by only one component in a cooperation. If a context is output, it can only appear in the outputting component. Otherwise, free contexts can appear in the interactions of all components.

$$\left| \begin{array}{l}
 \text{subordination} = \boxed{\text{subordinate}} \boxed{\text{process}} \boxed{;} \\
 \qquad \qquad \qquad | \boxed{\text{subordinate}} \boxed{(} \boxed{\text{replicator}} \boxed{)} \boxed{\text{process}} \boxed{;}
 \end{array} \right.$$

The components of a subordination start simultaneously and continue independently. A subordination terminates when all the components of the subordination have started.

A subordinate process may not contain references to free variables in its scope. Subordinate processes may not output a free context. A subordinate process terminates if it attempts to interact with a context whose scope has terminated or whose value has been output.

D.1.3 Replication

<i>replicator</i>	=	<i>name</i>	for	<i>count</i>	
			name		
			for	<i>count</i>	
			from	<i>base</i>	
			from	<i>base</i>	
			by	<i>step</i>	
<i>count</i>	=	<i>expression</i>			
<i>base</i>	=	<i>expression</i>			
<i>step</i>	=	<i>expression</i>			

D.1.4 Contexts – shared data structures

A context is a shared data structure, which may be an unordered bag, a stream or a singleton. The type of data in a context is defined by equivalence to those in the reference language.

For C programmers the most significant thing to observe is that pointers may not be used in contexts. C pointers are not meaningful things to exchange between processes since subordinates may not share the same address space. However, cooperating processes may share access to free pointers according to the rules for cooperation.

Indeed, put and get operations are designed to take care of data exchange by reference, since they manipulate pointers where they may and copy data where they may not.

Types

The types used in *Ease* operations exclude pointers and unions. However, no restriction is placed on the use of pointers within a process.

<i>type</i>	=	<i>data.type</i>			
			stream	<i>type</i>	
			[<i>expression</i>]
			reply	<i>data.type</i>	

$$\begin{array}{l}
 \left| \begin{array}{l}
 \text{data_type} \quad = \text{primitive_type} \\
 \quad \quad \quad | \text{array_type} \\
 \quad \quad \quad | \text{tuple_type} \\
 \text{primitive_type} \quad = \text{integer_type} \\
 \quad \quad \quad | \boxed{\text{unsigned}} \text{integer_type} \\
 \quad \quad \quad | \boxed{\text{signed}} \text{integer_type} \\
 \quad \quad \quad | \text{float_type} \\
 \boxed{\text{signed}} \text{integer_type} = \text{integer_type}
 \end{array} \right.
 \end{array}$$

The following equivalences define primitive types and syntax in C-with-Ease in terms of equivalence to those specified in the reference language.

$$\begin{array}{l}
 \text{integer_type} = \begin{array}{l}
 \boxed{\text{int}} \quad \quad \quad \equiv \text{INT} \\
 | \boxed{\text{char}} \quad \quad \quad \equiv \text{INT8} | \text{UNSIGNED INT8}^\dagger \\
 | \boxed{\text{short}} \quad \quad \quad \equiv \text{INT16} | \text{INT32}^\dagger \\
 | \boxed{\text{short int}} \quad \quad \quad \equiv \text{INT16} | \text{INT32}^\dagger \\
 | \boxed{\text{long}} \quad \quad \quad \equiv \text{INT32} | \text{INT64}^\dagger \\
 | \boxed{\text{long int}} \quad \quad \quad \equiv \text{INT32} | \text{INT64}^\dagger
 \end{array} \\
 \text{float_type} = \begin{array}{l}
 \boxed{\text{float}} \quad \quad \quad \equiv \text{FLOAT32} \\
 | \boxed{\text{double}} \quad \quad \quad \equiv \text{FLOAT64}
 \end{array}
 \end{array}$$

Those types marked by † are machine dependent. An implementation must state the true equivalence.

$$\left| \text{array_type} = \text{type} \boxed{[} \text{expression} \boxed{]} \right.$$

An array type is a homogeneous sequence of components of some type. The size of the sequence is specified by the associate expression, which must be of integer type.

These arrays are directly equivalent to arrays in the reference language.

$$\left| \begin{array}{l}
 \text{array_type} = \text{integer_type} \boxed{::} \boxed{[} \text{count} \boxed{]} \text{type} \\
 \text{count} \quad = \text{expression}
 \end{array} \right.$$

A counted array type is a *count* component of an integer type, followed by a homogeneous sequence of components of some type. The size of the sequence is bounded to be not greater than the value specified by the associated *count* expression.

Let e be an expression, I an integer type and t be some type, then the type $I :: [e]t$ is valid iff $e > 0$.

$$\left| \begin{array}{l} \text{tuple_type} = \boxed{\text{struct}} \boxed{\{ \langle _2, \text{type} \rangle \}} \end{array} \right.$$

Tuple types are equivalent to structurally equivalent ANSI C standard structures.

Type definitions

A name defined by a C type definition is valid only if the types in the definition are structurally compatible which those mentioned.

D.1.5 Context type definition

$$\left| \begin{array}{l} \text{definition} = \boxed{\text{context}} \text{ name type } \boxed{;} \\ \quad \quad \quad | \boxed{\text{context}} \text{ name } \boxed{\{ \langle _1 ; \text{type} \rangle \}} \\ \text{type} \quad \quad = \text{ name} \end{array} \right.$$

A context type definition defines a name for the specified context type. A context whose type is defined by a type definition is of the same type if their type has been defined in the same definition (i.e. name equivalence).

D.1.6 Context allocation

$$\left| \begin{array}{l} \text{allocation} = \boxed{\text{share}} \text{ type context } \boxed{;} \\ \quad \quad \quad | \text{ placement} \\ \text{context} \quad = \text{ name} \end{array} \right.$$

An allocation specifies a name for a context.

$$\begin{array}{l}
 \textit{placement} = \boxed{\text{share}} \textit{ type context } \boxed{\text{on}} \langle \textit{node} \rangle \\
 \quad \quad \quad | \boxed{\text{share}} \textit{ type context } \boxed{\text{on}} \textit{ node } \boxed{\text{at}} \textit{ address} \\
 \\
 \textit{node} \quad = \textit{ expression} \\
 \textit{address} = \textit{ expression}
 \end{array}$$

A placement allocates a context (which must be a singleton) on a node or group of nodes.

D.1.7 Interaction

$$\begin{array}{l}
 \textit{interaction} = \textit{input} | \textit{output} \\
 \\
 \textit{input} \quad = \textit{read} | \textit{get} \\
 \textit{output} \quad = \textit{write} | \textit{put} \\
 \\
 \textit{read} \quad = \boxed{\text{read}} \boxed{(} \textit{ context } \boxed{,} \textit{ variable } \boxed{)} \\
 \textit{write} \quad = \boxed{\text{write}} \boxed{(} \textit{ context } \boxed{,} \textit{ expression } \boxed{)} \\
 \\
 \textit{get} \quad = \boxed{\text{get}} \boxed{(} \textit{ context } \boxed{,} \textit{ name } \boxed{)} \\
 \textit{put} \quad = \boxed{\text{put}} \boxed{(} \textit{ context } \boxed{,} \textit{ name } \boxed{)}
 \end{array}$$

The interactions specified here are simple syntactic variants of those specified in the reference language.

There are four simple, symmetric, operations on contexts. They are

- write (c, e) – copies the value of the expression e to the context c.
- read (c, v) – copies a value from the context c to a variable v.
- put (c, n) – moves the value associated with the name n to the context c.
- get (c, n) – moves a value from the context c and binds it to the name n.

Write and read are copy operations. Put and get are binding operators.

The synchronization characteristics of the operations are similarly symmetric

- get and read block if data is not existent
- write and put are non-blocking.

Consider how these operations change the state of a program.

Write changes the state of a context, leaving the local state unchanged. Read changes the local state whilst leaving the context state unchanged.

Put changes both the context state and local state, i.e. subsequently the value associated with the variable name used in the operation is undefined. Get also changes both the context state and the local state, i.e. the value bound to the variable name used in the operation is removed from the context.

D.1.8 Resource

The construction of and interaction with resources has special requirements. To enable the simple and uniform view of resources in parallel and distributed environments, Ease provides *combinations*.

$$\left| \begin{array}{l} \textit{combination} = \textit{call} \mid \textit{reply} \\ \textit{call} = \text{call} \left(\textit{context}, \textit{expression}, \textit{name} \right) \\ \textit{reply} = \text{resource} \left(\textit{context}, \textit{name} \right) \\ \text{reply} \textit{function} ; \end{array} \right.$$

A *combination* provides guaranteed *call reply* semantics via some context. Access to system resources is provided by use of combinations.

A *call* behaves like an output and get in sequence.

A *reply* behaves like a get, process and output in sequence.

The behavior of a combination is described as the synchronization of the calling process and the resource process, where the output of the resource process in the call reply context is guaranteed to satisfy the input of the corresponding call.

D.1.9 Scope

The scope of a context allocated in a process is from the point of allocation to the end of that process. æ