Overview

Overview

Céu provides *Structured Synchronous Reactive Programming*, extending classical structured programming with two main functionalities:

- Event Handling:
  - An `await` statement to suspend a line of execution and wait for an input event from the environment.
  - An `emit` statement to signal an output event back to the environment.
- Concurrency:
  - A set of parallel constructs to compose concurrent lines of execution.

The lines of execution in Céu, known as *trails*, react all together to input events one after another, in discrete steps. An input event is broadcast to all active trails, which share the event as their unique and global time reference.

The program that follows blinks a LED every second and terminates on a button press:

```
input  void BUTTON;
output bool LED;
par/or do
    await BUTTON;
with
    loop do
        await 1s;
        emit LED(true);
        await 1s;
        emit LED(false);
    end
end
```

The synchronous concurrency model of Céu greatly diverges from multithreaded and actor-based models (e.g. *pthreads* and *erlang*). On the one hand, there is no real parallelism at the synchronous kernel of the language (i.e., no multi-core execution). On the other hand, accesses to shared variables among trails are deterministic and do not require synchronization primitives (i.e., *locks* or *queues*).

Céu provides static memory management based on lexical scopes and does not require a garbage collector.

Céu integrates safely with C, particularly when manipulating external resources (e.g., file handles). Programs can make native calls seamlessly while avoiding common pitfalls such as memory leaks and dangling pointers.

Céu is free software.

Environments

As a reactive language, Céu depends on an external host platform, known as an *environment*, which exposes `input` and `output` events programs can use.

An environment senses the world and broadcasts `input` events to programs. It also intercepts programs signalling `output` events to actuate in the world:

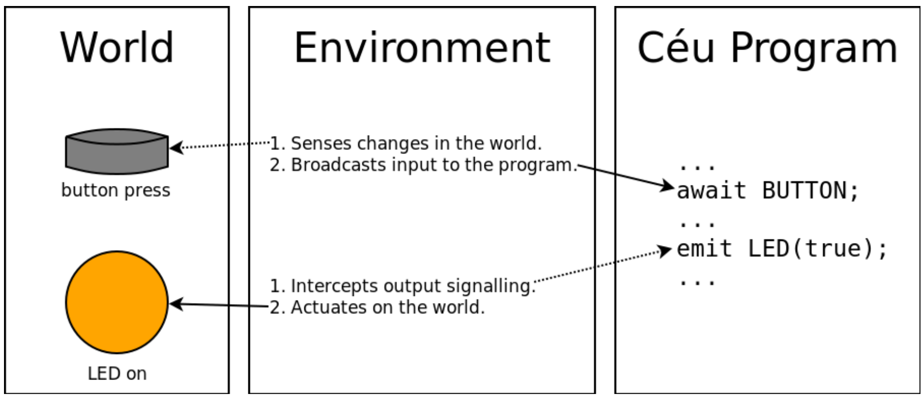

Figure 1:

As examples of typical environments, an embedded system may provide button input and LED output, and a video game engine may provide keyboard input and video output.

Synchronous Execution Model

Céu is grounded on a precise definition of *logical time* (as opposed to *physical*) as a discrete sequence of input events: a sequence because only a single input event is handled at a logical time; discrete because reactions to events are guaranteed to execute in bounded physical time (see Bounded Execution).

The execution model for Céu programs is as follows:

1. The program initiates the *boot reaction* from the first line of code in a single trail.
2. Active trails, one after another, execute until they await or terminate. This step is named a *reaction chain*, and always runs in bounded time. New trails can be created with parallel compositions.
3. The program goes idle.
4. On the occurrence of a new input event, *all* trails awaiting that event awake. It then goes to step 2.

The synchronous execution model of Céu is based on the hypothesis that reaction chains run *infinitely faster* in comparison to the rate of input events. A reaction chain, aka *external reaction*, is the set of computations that execute when an input event occurs. Conceptually, a program takes no time on step 2 and is always idle on step 3. In practice, if a new input event occurs while a reaction chain is running (step 2), it is enqueued to run in the next reaction. When multiple trails are active at a logical time (i.e. awaking from the same event), Céu schedules them in the order they appear in the program text. This policy is arbitrary, but provides a priority scheme for trails, and also ensures deterministic and reproducible execution for programs. At any time, at most one trail is executing.

The program and diagram below illustrate the behavior of the scheduler of Céu:

```
 1:  input void A, B, C;  // A, B, and C are input events
 2:  par/and do
 3:      // trail 1
 4:      <...>            // <...> represents non-awaiting statements
 5:      await A;
 6:      <...>
 7:  with
 8:      // trail 2
 9:      <...>
10:      await B;
11:      <...>
12:  with
13:      // trail 3
14:      <...>
15:      await A;
16:      <...>
17:      await B;
18:      par/and do
19:          // trail 3
20:          <...>
21:      with
22:          // trail 4
23:          <...>
24:      end
25:  end
```

The program starts in the boot reaction and forks into three trails. Respecting the lexical order of declaration for the trails, they are scheduled as follows (*t0* in the diagram):

- *trail-1* executes up to the `await A` (line 5);
- *trail-2* executes up to the `await B` (line 10);
- *trail-3* executes up to the `await A` (line 15).

As no other trails are pending, the reaction chain terminates and the scheduler remains idle until the event `A` occurs (*t1* in the diagram):

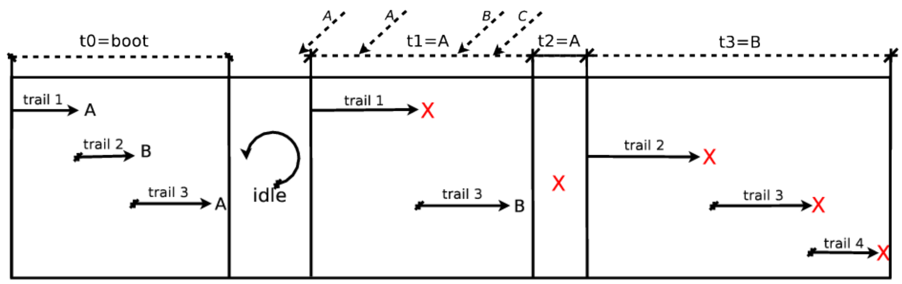

Figure 2:

- *trail-1* awakes, executes and terminates (line 6);
- *trail-2* remains suspended, as it is not awaiting `A`.
- *trail-3* executes up to `await B` (line 17).

During the reaction *t1*, new instances of events `A`, `B`, and `C` occur and are enqueued to be handled in the reactions in sequence. As `A` happened first, it is used in the next reaction. However, no trails are awaiting it, so an empty reaction chain takes place (*t2* in the diagram). The next reaction dequeues the event `B` (*t3* in the diagram):

- *trail-2* awakes, executes and terminates;
- *trail-3* splits in two and they both terminate immediately.

Since a `par/and` rejoins after all trails terminate, the program also terminates and does not react to the pending event `C`.

Note that each step in the logical time line (*t0*, *t1*, etc.) is identified by the unique occurring event. Inside a reaction, trails only react to the same shared global event (or remain suspended).

Parallel Compositions and Abortion

The use of trails in parallel allows programs to wait for multiple events at the same time. Céu supports three kinds of parallel compositions that differ in how they rejoin and proceed to the statement in sequence:

1. a `par/and` rejoins after all trails in parallel terminate;
2. a `par/or` rejoins after any trail in parallel terminates, aborting all other trails automatically;
3. a `par` never rejoins, even if all trails terminate.

As mentioned in the introduction and emphasized in the execution model, trails in parallel do not execute with real parallelism. Therefore, parallel compositions support *awaiting in parallel*, rather than *executing in parallel*.

Bounded Execution

Reaction chains must run in bounded time to guarantee that programs are responsive and can handle incoming input events. For this reason, Céu requires every path inside the body of a `loop` statement to contain at least one `await` or `break` statement. This prevents *tight loops*, i.e., unbounded loops that do not await.

In the example below, the true branch of the `if` may never execute, resulting in a tight loop when the condition is false:

```
loop do
    if <cond> then
        break;
    end
end
```

Céu warns about tight loops in programs at compile time. For time-consuming algorithms that require unrestricted loops (e.g., cryptography, image processing), Céu provides Asynchronous Execution.

Deterministic Execution

```
TODO (shared memory + deterministic scheduler + optional static
analysis)
```

Internal Reactions

Céu supports inter-trail communication through `await` and `emit` statements for *internal events*. A trail can `await` an internal event to suspend it. Then, another trail can `emit` and broadcast an event, awaking all trails awaiting that event.

Unlike input events, multiple internal events can coexist during an external reaction. An `emit` starts a new *internal reaction* in the program which relies on a runtime stack:

1. The `emit` suspends the current trail and its continuation is pushed into the stack (i.e., the statement in sequence with the `emit`).
2. All trails awaiting the emitted event awake and execute in sequence (see `rule 2` for external reactions). If an awaking trail emits another internal event, a nested internal reaction starts with `rule 1`.
3. The top of stack is popped and the last emitting trail resumes execution from its continuation.

Example:

```
par/and do      // trail 1
    await e;
```

```
3:      emit f;
4:  with            // trail 2
5:      await f;
6:  with            // trail 3
8:      emit e;
9:  end
```

The `emit e` in *trail-3* (line 7) starts an internal reaction that awakes the `await e` in *trail-1* (line 2). Then, the `emit f` (line 3) starts another internal reaction that awakes the `await f` in *trail-2* (line 5). *Trail-2* terminates and the `emit f` resumes in *trail-1*. *Trail-1* terminates and the `emit e` resumes in *trail-3*. *Trail-3* terminates. Finally, the `par/and` rejoins and the program terminates.

Lexical Rules

Lexical Rules

Keywords

Keywords in Céu are reserved names that cannot be used as identifiers (e.g., variable names):

`and`	`as`	`async`	`atomic`	`await`
`break`	`call`	`code`	`const`	`continue`
`data`	`deterministic`	`do`	`dynamic`	`else`
`emit`	`end`	`escape`	`event`	`every`
`false`	`finalize`	`FOREVER`	`hold`	`if`
`in`	`input`	`is`	`isr`	`kill`
`lock`	`loop`	`lua`	`native`	`new`
`nohold`	`not`	`nothing`	`nil`	`null`
`or`	`outer`	`output`	`par`	`pause`
`plain`	`pool`	`pos`	`pre`	`pure`
`recursive`	`request`	`resume`	`sizeof`	`spawn`
`static`	`then`	`thread`	`tight`	`traverse`

```
true          until         val           var           vector

watching      with          bool          byte          f32

f64           float         int           s16           s32

s64           s8            ssize         u16           u32

u64           u8            uint          usize         void
```

Identifiers

Céu uses identifiers to refer to *types* (`ID_type`), *variables* (`ID_int`), *vectors* (`ID_int`), *pools* (`ID_int`), *internal events* (`ID_int`), *external events* (`ID_ext`), *code abstractions* (`ID_abs`), *data abstractions* (`ID_abs`), *fields* (`ID_field`), *native symbols* (`ID_nat`), and *block labels* (`ID_int`).

```
ID       ::= [a-z, A-Z, 0-9, _]+
ID_int   ::= ID             // ID beginning with lowercase
ID_ext   ::= ID             // ID all in uppercase, not beginning with digit
ID_abs   ::= ID {`.´ ID}    // IDs beginning with uppercase, containining at least one lower
ID_field ::= ID             // ID not beginning with digit
ID_nat   ::= ID             // ID beginning with underscore

ID_type  ::= ( ID_nat | ID_abs
             | void  | bool  | byte
             | f32   | f64   | float
             | s8    | s16   | s32   | s64
             | u8    | u16   | u32   | u64
             | int   | uint  | ssize | usize )
```

Declarations for `code` and `data` create new types which can be used as type identifiers.

Examples:

```
var int a;                    // "a" is a variable, "int" is a type
emit e;                       // "e" is an internal event
await E;                      // "E" is an external input event
spawn Move();                 // "Move" is a code abstraction and a type
var Rect r;                   // "Rect" is a data abstraction and a type
return r.width;               // "width" is a field
_printf("hello world!\n");    // "_printf" is a native symbol
```

Literals

Céu supports literals for *booleans*, *integers*, *floats*, *strings*, and *null pointers*.

Booleans

The boolean type has only two possible values: `true` and `false`.

Integers

Céu supports decimal and hexadecimal integers:

- Decimals: a sequence of digits (i.e., `[0-9]+`).
- Hexadecimals: a sequence of hexadecimal digits (i.e., `[0-9, a-f, A-F]+`) prefixed by 0x.

Examples:

```
// both are equal to the decimal 127
v = 127;    // decimal
v = 0x7F;   // hexadecimal
```

Floats

```
TODO (like C)
```

Strings

A sequence of characters surrounded by `"` is converted into a *null-terminated string*, just like in C:

Example:

```
_printf("Hello World!\n");
```

Null pointer

```
TODO (like C)
```

Comments

Céu provides C-style comments:

- Single-line comments begin with `//` and run to end of the line.
- Multi-line comments use `/*` and `*/` as delimiters. Multi-line comments can be nested by using a different number of `*` as delimiters.

Examples:

```
var int a;    // this is a single-line comment

/** comments a block that contains comments

var int a;
/* this is a nested multi-line comment
a = 1;
*/

**/
```

Types

Types

Céu is statically typed, requiring all variables, events, and other entities to be declared before they are used in programs.

A type is composed of a type identifier, followed by an optional sequence of pointer modifiers `&&`, followed by an optional option modifier `?`:

```
Type ::= ID_type {`&&´} [`?´]
```

Examples:

```
var   u8     v;    // "v" is of 8-bit unsigned integer type
var   _rect  r;    // "r" is of external native type "rect"
var   Tree   t;    // "t" is a data of type "Tree"
var   int?   ret;  // "ret" is either unset or is of integer type
input byte&& RECV; // "RECV" is an input event carrying a pointer to a "byte"
```

Primitives

Céu has the following primitive types:

```
void               // void type
bool               // boolean type
byte               // 1-byte type
int      uint      // platform dependent signed and unsigned integer
s8       u8        // signed and unsigned  8-bit integers
s16      u16       // signed and unsigned 16-bit integers
s32      u32       // signed and unsigned 32-bit integers
s64      u64       // signed and unsigned 64-bit integers
float              // platform dependent float
f32      f64       // 32-bit and 64-bit floats
ssize    usize     // signed and unsigned size types
```

Natives

Types defined externally in C can be prefixed by `_` to be used in Céu programs.

Example:

```
var _message_t msg;      // "message_t" is a C type defined in an external library
```

Native types support modifiers to provide additional information to the compiler.

Abstractions

See Abstractions.

Modifiers

Types can be suffixed with the pointer modifier `&&` and the option modifier `?`.

Pointer

`TODO (like in C)`

`TODO cannot cross yielding statements`

Option

`TODO (like "Maybe")`

`TODO: nil`

Storage Entities

Storage Entities

Storage entities represent all objects that are stored in memory during execution. Céu supports *variables*, *vectors*, *events* (external and internal), and *pools* as entity classes.

An entity declaration consists of an entity class, a type, and an identifier.

Examples:

```
var       int    v;     // "v" is a variable of type "int"
vector[9] byte   buf;   // "buf" is a vector with at most 9 values of type "byte"
input     void&& A;     // "A" is an input event that carries values of type "void&&"
event     bool   e;     // "e" is an internal event that carries values of type "bool"
```

```
pool[]    Anim   anims; // "anims" is a dynamic "pool" for instances of type "Anim"
```

A declaration binds the identifier with a memory location that holds values of the associated type.

Lexical Scope

Storage entities have lexical scope, i.e., they are visible only in the block in which they are declared.

The lifetime of entities, which is the period between allocation and deallocation in memory, is also limited to the scope of the enclosing block. However, individual elements inside *vector* and *pool* entities have dynamic lifetime, but which never outlive the scope of the declaration.

Entity Classes

Variables

A variable in Céu holds a value of a declared type that may vary during program execution. The value of a variable can be read in expressions or written in assignments. The current value of a variable is preserved until the next assignment, during its whole lifetime.

Example:

```
var int v = _;  // empty initializaton
par/and do
    v = 1;      // write access
with
    v = 2;      // write access
end
escape v;       // read access (yields 2)
```

Vectors

A vector In Céu is a dynamic and contiguous collection of elements of the same type.

A vector declaration specifies its type and maximum number of elements (possibly unlimited). The current length of a vector is dynamic and can be accessed through the operator `$`.

Individual elements of a vector can be accessed through an index starting from `0`. Céu generates an error for out-of-bounds vector accesses.

Example:

```
vector[9] byte buf = [1,2,3];   // write access
buf = buf .. [4];               // write access
escape buf[1];                  // read access (yields 2)
```

Events

Events account for the reactive nature of Céu. Programs manipulate events through the `await` and `emit` statements. An `await` halts the running trail until the specified event occurs. An event occurrence is broadcast to the whole program and awakes trails awaiting that event to resume execution.

Unlike all other entity classes, the value of an event is ephemeral and does not persist after a reaction terminates. For this reason, an event identifier is not a variable: values can only be communicated through `emit` and `await` statements. A declaration includes the type of value the occurring event carries.

Note: void is a valid type for signal-only events.

Example:

```
input  void I;           // "I" is an input event that carries no values
output int  O;           // "O" is an output event that carries values of type "int"
event  int  e;           // "e" is an internal event that carries values of type "int"
par/and do
    await I;             // awakes when "I" occurs
    emit e(10);          // broadcasts "e" passing 10, awakes the "await" below
with
    var int v = await e; // awaits "e" assigning the received value to "v"
    emit O(v);           // emits "O" back to the environment passing "v"
end
```

As described in Internal Reactions, Céu supports external and internal events with different behavior.

External Events

External events are used as interfaces between programs and devices from the real world:

- *input events* represent input devices such as sensor, button, mouse, etc.
- *output events* represent output devices such as LED, motor, screen, etc.

The availability of external events depends on the environment in use.

Programs can `emit` output events and `await` input events.

Internal Events

Internal events, unlike external events, do not represent real devices and are defined by the programmer. Internal events serve as signalling and communication mechanisms among trails in a program.

Programs can `emit` and `await` internal events.

Pools

A pool is a dynamic container to hold running code abstractions.

A pool declaration specifies the type of the abstraction and maximum number of concurrent instances (possibly unlimited). Individual elements of pools can only be accessed through iterators. New elements are created with `spawn` and are removed automatically when the code execution terminates.

Example:

```
code/await Anim (void) => void do       // defines the "Anim" code abstraction
    <...>                               // body of "Anim"
end
pool[] Anim ms;                         // declares an unlimited container for "Anim" instar
loop i in [0->10[ do
    spawn Anim() in ms;                 // creates 10 instances of "Anim" into "ms"
end
```

When a pool declaration goes out of scope, all running code abstractions are automatically aborted.

Locations

A location (aka *l-value*) is a path to a memory location holding a storage entity (`ID_int`) or a native symbol (`ID_nat`).

Locations appear in assignments, event manipulation, iterators, and expressions.

The list that follows summarizes all valid locations:

- storage entity: variable, vector, internal event (but not external), or pool
- native expression or symbol
- data field (which are storage entities)
- vector index
- vector length `$`
- pointer dereferencing `*`
- option unwrapping `!`

Locations are detailed in Locations and Expressions.

Examples:

```
emit e(1);          // "e" is an internal event
_UDR = 10;          // "_UDR" is a native symbol
person.age = 70;    // "age" is a variable in "person"
vec[0] = $vec;      // "vec[0]" is a vector index
$vec = 1;           // "$vec" is a vector length
*ptr = 1;           // "ptr" is a pointer to a variable
a! = 1;             // "a" is of an option type
```

References

Céu supports *aliases* and *pointers* as references to entities, aka *strong* and *weak* references, respectively.

An alias is an alternate view for an entity: after the entity and alias are bounded, they are indistinguishable.

A pointer is a value that is the address of an entity, providing indirect access to it.

As an analogy with a person's identity, a family nickname referring to a person is an alias; a job position referring to a person is a pointer.

Aliases

Céu support aliases to all storage entity classes, except external events and pointer types. Céu also supports option variable aliases which are aliases that may remain or become unassigned.

An alias is declared by suffixing the entity class with the modifier `&` and is acquired by prefixing an entity with the operator `&`.

An alias must have a narrower scope than the entity it refers to. The assignment to the alias is immutable and must occur between its declaration and first access or next yielding statement.

Example:

```
var  int v = 0;
var& int a = &v;        // "a" is an alias to "v"
...
a = 1;                  // "a" and "v" are indistinguishable
_printf("%d\n", v);     // prints 1
```

An option variable alias, declared as `var&?`, serves two purposes:

- Map a native resource to a variable in Céu. The alias is acquired by prefixing the associated native call with the operator `&`. Since the allocation may fail, the alias may remain unassigned.

- Track the lifetime of a variable. The alias is acquired by prefixing the associated variable with the operator `&`. Since the tracked variable may go out of scope, the alias may become unassigned.

Accesses to option variable aliases must always use option checking or unwrapping.

Examples:

```
var&? _FILE f = &_fopen(<...>) finalize with
                    _fclose(f);
                end;
if f? then
    <...>   // "f" is assigned
else
    <...>   // "f" is not assigned
end
```

```
var&? int x;
do
    var int y = 10;
    x = &y;
    _printf("%d\n", x!);    // prints 10
end
_printf("%d\n", x!);        // error!
```

Pointers

A pointer is declared by suffixing the type with the modifier `&&` and is acquired by prefixing an entity with the operator `&&`. Applying the operator `*` to a pointer provides indirect access to its referenced entity.

Example:

```
var int   v = 0;
var int&& p = &&v;      // "p" holds a pointer to "v"
...
*p = 1;                 // "p" provides indirect access to "v"
_printf("%d\n", v);     // prints 1
```

The following restrictions apply to pointers in Céu:

- No support for pointers to events, vectors, or pools (only variables).
- A pointer is only accessible between its declaration and the next yielding statement.

Statements

Statements

A program in Céu is a sequence of statements delimited by an implicit enclosing block:

```
Program ::= Block
Block   ::= {Stmt `;´} {`;´}
```

Note: statements terminated with the `end` keyword do not require a terminating semicolon.

Nothing

`nothing` is an innocuous statement:

```
Nothing ::= nothing
```

Blocks

A `Block` creates a new lexical scope for storage entities and abstractions, which are visible only for statements inside the block.

Compound statements (e.g. *do-end*, *if-then-else*, *loops*, etc.) create new blocks and can be nested to an arbitrary level.

`do-end` and `escape`

The `do-end` statement creates an explicit block with an optional identifier following the symbol `/`. The `escape` statement aborts the deepest enclosing `do-end` matching its identifier:

```
Do ::= do [`/´(ID_int|`_´)] [`(´ [LIST(ID_int)] `)´]
           Block
       end

Escape ::= escape [`/´ID_int] [Exp]
```

The neutral identifier `_` which is guaranteed not to match any `escape` statement.

A `do-end` also supports an optional list of identifiers in parenthesis which restricts the visible variables inside the block to those matching the list.

A `do-end` can be assigned to a variable whose type must be matched by nested `escape` statements. The whole block evaluates to the value of a reached `escape`.

If the variable is of option type, the `do-end` is allowed to terminate without an `escape`, otherwise it raises a runtime error.

Programs have an implicit enclosing `do-end` that assigns to a *program status variable* of type `int` whose meaning is platform dependent.

Examples:

```
do
    do/a
        do/_
            escape;     // matches line 1
        end
        escape/a;       // matches line 2
    end
end
```

“‘ceu var int a; var int b; do (a) a = 1; b = 2; // “b” is not visible end

```
var int? v =
    do
        if <cnd> then
            escape 10;  // assigns 10 to "v"
        else
            nothing;    // "v" remains unassigned
        end
    end;

escape 0;               // program terminates with a status value of 0
```

`pre-do-end`

The `pre-do-end` statement prepends its statements in the beginning of the program:

```
Pre_Do ::= pre do
               Block
           end
```

All `pre-do-end` statements are concatenated together in the order they appear and moved to the beginning of the top-level block, before all other statements.

Declarations

A declaration introduces a storage entity to the enclosing block. All declarations are subject to lexical scope.

Céu supports variables, vectors, pools, internal events, and external events:

```
Var  ::= var [`&´|`&?´] [`/dynamic´|`/nohold´] Type ID_int [`=´ Sources]
Vec  ::= vector [`&´] `[´ [Exp] `]´ Type ID_int [`=´ Sources]
Pool ::= pool [`&´] `[´ [Exp] `]´ Type ID_int [`=´ Sources]
Int  ::= event [`&´] (Type | `(´ LIST(Type) `)´) ID_int [`=´ Sources]

Ext  ::= input  (Type | `(´ LIST(Type) `)´) ID_ext
      |  output (Type | `(´ LIST([`&´] Type) `)´) ID_ext

Sources ::= /* (see "Assignments") */
```

Most declarations support an initialization assignment.

Variables

A variable declaration has an associated type and can be optionally initialized. Declarations can also be aliases or option aliases.

Examples:

```
var  int v = 10;    // "v" is an integer variable initialized to 10
var  int a=0, b=3;  // "a" and "b" are integer variables initialized to 0 and 3
var& int z = &v;    // "z" is an alias to "v"
```

Vectors

A vector declaration specifies a dimension between brackets, an associated type and can be optionally initialized. Declarations can also be aliases.

Examples:

```
var int n = 10;
vector[10] int vs1 = [];    // "vs1" is a static vector of 10 elements max
vector[n]  int vs2 = [];    // "vs2" is a dynamic vector of 10 elements max
vector[]   int vs3 = [];    // "vs3" is an unbounded vector
vector&[]  int vs4 = &vs1;  // "vs4" is an alias to "vs1"
```

Pools

A pool declaration specifies a dimension and an associated type. Declarations for pools can also be aliases. Only in this case they can be initialized.

The expression between the brackets specifies the dimension of the pool.

Examples:

```
code/await Play (...) do ... end
pool[10] Play plays;        // "plays" is a static pool of 10 elements max
pool&[]  Play a = &plays;   // "a" is an alias to "plays"
```

TODO: data

Dimension

Declarations for vectors or pools require an expression between brackets to specify a dimension as follows:

- *constant expression*: Maximum number of elements is fixed and space is statically pre-allocated.
- *variable expression*: Maximum number of elements is fixed but space is dynamically allocated. The expression is evaulated once at declaration time.
- *omitted*: Maximum number of elements is unbounded and space is dynamically allocated. The space for dynamic dimensions grow and shrink automatically.

Events

An event declaration specifies a type for the values it carries when occurring. It can be also a list of types if the event communicates multiple values.

External Events

Examples:

```
input  void A,B;        // "A" and "B" are input events carrying no values
output int  MY_EVT;     // "MY_EVT" is an output event carrying integer values
input (int,byte&&) BUF; // "BUF" is an input event carrying an "(int,byte&&)" pair
```

TODO: output &

Internal Events

Declarations for internal events can also be aliases. Only in this case they can be initialized.

Examples:

```
event  void a,b;        // "a" and "b" are internal events carrying no values
event& void z = &a;     // "z" is an alias to event "a"
event (int,int) c;      // "c" is a internal event carrying an "(int,int)" pair
```

Assignments

An assignment associates the statement or expression at the right side of the symbol = with the location(s) at the left side:

```
Assignment ::= (Loc | `(´ LIST(Loc|`_´) `)´) `=´ Sources

Sources ::= ( Do
            | Emit_Ext
            | Await
            | Watching
            | Thread
            | Lua_State
            | Lua_Stmts
            | Code_Await
            | Code_Spawn
            | Vec_Cons
            | Data_Cons
            | Exp
            | `nil´
            | `_´ )
```

Céu supports the following constructs as assignment sources:

- `do-end` block
- external emit
- await
- watching statement
- thread
- lua state
- lua statement
- code await
- code spawn
- vector length & constructor
- data constructor
- expression
- the option `nil` value
- the anonymous identifier `_`

The option `nil` value unsets an option variable.

The anonymous identifier makes the assignment innocuous.

```
TODO: required for uninitialized variables
```

Copy Assignment

A *copy assignment* evaluates the statement or expression at the right side and copies the result(s) to the location(s) at the left side.

Alias Assignment

An *alias assignment*, aka *binding*, makes the location at the left side to be an alias to the expression at the right side.

The right side of a binding is always prefixed by the operator `&`.

Event Handling

Await

The `await` statement halts the running trail until the specified event occurs. The event can be an input event, an internal event, a timer, a pausing event, or forever (i.e., never awakes):

```
Await ::= await (ID_ext | Loc) [until Exp]      /* events and option aliases */
       |  await (WCLOCKK|WCLOCKE)               /* timers */
       |  await (pause|resume)                  /* pausing events */
       |  await FOREVER                         /* forever */
```

Examples:

```
await A;                  // awaits the input event "A"
await a until v==10;      // awaits the internal event "a" until the condition is satisfied

await 1min10s30ms100us;   // awaits the specified time
await (t)ms;              // awaits the current value of the variable "t" in milliseconds

await FOREVER;            // awaits forever
```

An `await` evaluates to zero or more values which can be captured with an optional assignment.

Event

The `await` statement for events halts the running trail until the specified input event or internal event occurs. The `await` evaluates to a value of the type of the event.

The optional clause `until` tests an awaking condition. The condition can use the returned value from the `await`. It expands to a `loop` as follows:

```
loop do
    <ret> = await <evt>;
    if <Exp> then   // <Exp> can use <ret>
        break;
    end
end
```

Examples:

```
input int E;                     // "E" is an input event carrying "int" values
var int v = await E until v>10; // assigns occurring "E" to "v", awaking only when "v>10"

event (bool,int) e;              // "e" is an internal event carrying "(bool,int)" pairs
var bool v1;
var int  v2;
(v1,v2) = await e;               // awakes on "e" and assigns its values to "v1" and "v2"
```

Option Alias

The `await` statement for option variable aliases halts the running trail until the specified alias goes out of scope.

The `await` evaluates to no value.

Example:

```
var&? int x;
spawn Code() -> (&x);   // "x" is bounded to a variable inside "Code"
await x;                // awakes when the spawned "Code" terminates
```

Timer

The `await` statement for timers halts the running trail until the specified timer expires:

- `WCLOCKK` specifies a constant timer expressed as a sequence of value/unit pairs.
- `WCLOCKE` specifies an integer expression in parenthesis followed by a single unit of time.

The `await` evaluates to a value of type `s32` and is the *residual delta time (`dt`)* measured in microseconds: the difference between the actual elapsed time and the requested time. The residual `dt` is always greater than or equal to 0.

If a program awaits timers in sequence (or in a `loop`), the residual `dt` from the preceding timer is reduced from the timer in sequence.

Examples:

```
var int t = <...>;
await (t)ms;                // awakes after "t" milliseconds

var int dt = await 100us;   // if 1000us elapses, then dt=900us (1000-100)
await 100us;                // since dt=900, this timer is also expired, now dt=800us (900-1
await 1ms;                  // this timer only awaits 200us (1000-800)
```

Pausing

Pausing events are dicussed in Pausing.

`FOREVER`

The `await` statement for `FOREVER` halts the running trail forever. It cannot be used in assignments because it never evaluates to anything.

Example:

```
if v==10 then
    await FOREVER;  // this trail never awakes if condition is true
end
```

Emit

The `emit` statement broadcasts an event to the whole program. The event can be an external event, an internal event, or a timer:

```
Emit_Int ::= emit Loc [`(´ [LIST(Exp)] `)´]
Emit_Ext ::= emit ID_ext [`(´ [LIST(Exp)] `)´]
          |  emit (WCLOCKK|WCLOCKE)
```

Examples:

```
emit A;         // emits the output event `A` of type "void"
emit a(1);      // emits the internal event `a` of type "int"

emit 1s;        // emits the specified time
emit (t)ms;     // emits the current value of the variable `t` in milliseconds
```

Events

The `emit` statement for events expects the arguments to match the event type.

An `emit` to an input or timer event can only occur inside asynchronous blocks.

An `emit` to an output event is also an expression that evaluates to a value of type `s32` and can be captured with an optional assignment (its meaning is platform dependent).

An `emit` to an internal event starts a new internal reaction.

Examples:

```
input int I;
async do
    emit I(10);         // broadcasts "I" to the application itself, passing "10"
end

output void O;
var int ret = emit O(); // outputs "O" to the environment and captures the result

event (int,int) e;
```

```
emit e(1,2);            // broadcasts "e" passing a pair of "int" values
```

Timer

The `emit` statement for timers expects a timer expression.

Like input events, time can only be emitted inside asynchronous blocks.

Examples:

```
async do
    emit 1s;    // broadcasts "1s" to the application itself
end
```

Lock

TODO

Conditional

The `if-then-else` statement provides conditional execution in Céu:

```
If ::= if Exp then
           Block
       { else/if Exp then
           Block }
       [ else
           Block ]
       end
```

Each condition `Exp` is tested in sequence, first for the `if` clause and then for each of the optional `else/if` clauses. On the first condition that evaluates to `true`, the `Block` following it executes. If all conditions fail, the optional `else` clause executes.

All conditions must evaluate to a value of type `bool`.

Loops

Céu supports simple loops, numeric iterators, event iterators, and pool iterators:

```
Loop ::=
      /* simple loop */
        loop [`/´Exp] do
            Block
        end
```

```
       /* numeric iterator */
       | loop [`/´Exp] Numeric do
             Block
         end

       /* event iterator */
       | every [(Loc | `(´ LIST(Loc|`_´) `)´) in] (ID_ext|Loc|WCLOCKK|WCLOCKE) do
             Block
         end

       /* pool iterator */
       | loop [`/´Exp] (ID_int|`_´) in Loc do
             Block
         end

Break    ::= break [`/´ID_int]
Continue ::= continue [`/´ID_int]

Numeric ::= /* (see "Numeric Iterators") */
```

The body of a loop `Block` executes an arbitrary number of times, depending on the conditions imposed by each kind of loop.

Except for the `every` iterator, all loops support an optional constant expression '/´Exp that limits the maximum number of iterations to avoid infinite execution. If the number of iterations reaches the limit, a runtime error occurs.

`break` and `continue`

The `break` statement aborts the deepest enclosing loop.

The `continue` statement aborts the body of the deepest enclosing loop and restarts it in the next iteration.

The optional modifier '/´ID_int in both statements only applies to numeric iterators.

Simple Loop

The simple `loop-do-end` statement executes its body forever. The only way to terminate a simple loop is with the `break` statement.

Examples:

```
// blinks a LED with a frequency of 1s forever
loop do
    emit LED(1);
    await 1s;
```

```
    emit LED(0);
    await 1s;
end
```

```
loop do
    loop do
        if <cnd-1> then
            break;      // aborts the loop at line 2 if <cnd-1> is satisfied
        end
    end
    if <cnd-2> then
        continue;       // restarts the loop at line 1 if <cnd-2> is satisfied
    end
end
```

Numeric Iterator

The numeric loop executes its body a fixed number of times based on a numeric range for a control variable:

```
Numeric ::= (`_´|ID_int) in [ (`[´ | `]´)
                                   ( (     Exp `->´ (`_´|Exp))
                                   | (`_´|Exp) `<-´ Exp      ) )
                               (`[´ | `]´) [`,´ Exp] ]
```

The control variable assumes the values specified in the interval, one by one, for each iteration of the loop body:

- **control variable:** `ID_int` is a read-only variable of a numeric type. Alternatively, the special anonymous identifier `_` can be used if the body of the loop does not access the variable.
- **interval:** Specifies a direction, endpoints with open or closed modifiers, and a step.
  - **direction**:
    - `->`: Starts from the endpoint `Exp` on the left increasing towards `Exp` on the right.
    - `<-`: Starts from the endpoint `Exp` on the right decreasing towards `Exp` on the left. Typically, the value on the left is smaller or equal to the value on the right.
  - **endpoints**: `[Exp` and `Exp]` are closed intervals which include `Exp` as the endpoints; `]Exp` and `Exp[` are open intervals which exclude `Exp` as the endpoints. Alternatively, the finishing endpoint may be `_` which means that the interval goes towards infinite.
  - **step**: An optional positive number added or subtracted towards the limit. If the step is omitted, it assumes the value `1`. If the direction is `->`, the step is added, otherwise it is subtracted.

  If the interval is not specified, it assumes the default `[0 -> _]`.

A numeric iterator executes as follows:

- **initialization:** The starting endpoint is assigned to the control variable. If the starting enpoint is open, the control variable accumulates a step immediately.
- **iteration:**
    1. **limit check:** If the control variable crossed the finishing endpoint, the loop terminates.
    2. **body execution:** The loop body executes.
    3. **step** Applies a step to the control variable. Goto step `1`.

The `break` and `continue` statements inside numeric iterators accept an optional modifier ‘/´ID_int to affect the enclosing loop matching the control variable.

Examples:

```
// prints "i=0", "i=1", ...
var int i;
loop i do
    _printf("i=%d\n", i);
end
```

```
// awaits 1s and prints "Hello World!" 10 times
loop _ in [0 -> 10[ do
    await 1s;
    _printf("Hello World!\n");
end
```

```
var int i;
loop i do
    var int j;
    loop j do
        if <cnd-1> then
            continue/i;         // continues the loop at line 1
        else/if <cnd-2> then
            break/j;            // breaks the loop at line 4
        end
    end
end
```

Note : the runtime asserts that the step is a positive number and that the control variable does not overflow.

Event Iterator

The `every` statement iterates over an event continuously, executing its body whenever the event occurs.

The event can be an external or internal event or a timer.

The optional assignment to a variable (or list of variables) stores the carrying value(s) of the event.

An `every` expands to a `loop` as illustrated below:

```
every <vars> in <event> do
    <body>
end
```

is equivalent to

```
loop do
    <vars> = await <event>;
    <body>
end
```

However, the body of an `every` cannot contain synchronous control statements, ensuring that no occurrences of the specified event are ever missed.

Examples:

```
every 1s do
    _printf("Hello World!\n");      // prints the "Hello World!" message on every second
end

event (bool,int) e;
var bool cnd;
var int  v;
every (cnd,v) in e do
    if not cnd then
        break;                      // terminates when the received "cnd" is false
    else
        _printf("v = %d\n", v);     // prints the received "v" otherwise
    end
end
```

Pool Iterator

TODO

Parallel Compositions

```
Pars ::= (par | par/and | par/or) do
             Block
         with
             Block
         { with
```

```
            Block }
        end

Spawn ::= spawn [`(´ [LIST(ID_int)] `)´] do
              Block
          end

Watching ::= watching LIST(ID_ext|Loc|WCLOCKK|WCLOCKE|Code_Cons_Init) do
                 Block
             end
```

The parallel statements `par/and`, `par/or`, and `par` fork the running trail in multiple others. They differ only on how trails rejoin and terminate the composition.

The `spawn` statement starts to execute a block in parallel with the enclosing block.

The `watching` statement executes a block and terminates when one of its specified events occur.

See also Parallel Compositions and Abortion.

par

The `par` statement never rejoins.

Examples:

```
// reacts continuously to "1s" and "KEY_PRESSED" and never terminates
input void KEY_PRESSED;
par do
    every 1s do
        <...>           // does something every "1s"
    end
with
    every KEY_PRESSED do
        <...>           // does something every "KEY_PRESSED"
    end
end
```

par/and

The `par/and` statement stands for *parallel-and* and rejoins when all nested trails terminate.

Examples:

```
// reacts once to "1s" and "KEY_PRESSED" and terminates
input void KEY_PRESSED;
```

```
par/and do
    await 1s;
    <...>               // does something after "1s"
with
    await KEY_PRESSED;
    <...>               // does something after "KEY_PRESSED"
end
```

par/or

The `par/or` statement stands for *parallel-or* and rejoins when any of the trails terminate, aborting all other trails.

Examples:

```
// reacts once to `1s` or `KEY_PRESSED` and terminates
input void KEY_PRESSED;
par/or do
    await 1s;
    <...>               // does something after "1s"
with
    await KEY_PRESSED;
    <...>               // does something after "KEY_PRESSED"
end
```

spawn

The `spawn` statement starts to execute a block in parallel with the enclosing block. When the enclosing block terminates, the spawned block is aborted.

Like a `do-end` block, a `spawn` also supports an optional list of identifiers in parenthesis which restricts the visible variables inside the block to those matching the list.

Examples:

```
spawn do
    every 1s do
        <...>       // does something every "1s"...
    end
end

<...>               // ...in parallel with whatever comes next
```

watching

A `watching` expands to a `par/or` with *n+1* trails: one to await each of the listed events, and one to execute its body, i.e.:

```
watching <e1>,<e2>,... do
    <body>
end
```

expands to

```
par/or do
    await <e1>;
with
    await <e2>;
with
    ...
with
    <body>
end
```

The `watching` statement accepts a list of events and terminates when any of them occur. It evaluates to what the occurring event value(s), which can be captured with an optional assignment.

Examples:

```
// reacts continuously to "KEY_PRESSED" during "1s"
input void KEY_PRESSED;
watching 1s do
    every KEY_PRESSED do
        <...>           // does something every "KEY_PRESSED"
    end
end
```

Pausing

The `pause/if` statement controls if its body should temporarily stop to react to events:

```
Pause_If ::= pause/if (Loc|ID_ext) do
                 Block
             end

Pause_Await ::= await (pause|resume)
```

A `pause/if` specifies a pausing event of type `bool` which, when emitted, toggles between pausing (`true`) and resuming (`false`) reactions for its body.

When its body terminates, the whole `pause/if` terminates and proceeds to the statement in sequence.

In transition instants, the body can react to the special `pause` and `resume` events before the corresponding state applies.

TODO: finalize/pause/resume

Examples:

```
event bool e;
pause/if e do       // pauses/resumes the nested body on each "e"
    every 1s do
        <...>       // does something every "1s"
    end
end
```

```
event bool e;
pause/if e do               // pauses/resumes the nested body on each "e"
    <...>
        loop do
            await pause;
            <...>           // does something before pausing
            await resume;
            <...>           // does something before resuming
        end
    <...>
end
```

Asynchronous Execution

Asynchronous execution allow programs to departure from the rigorous synchronous model and preform computations under separate scheduling rules.

Céu supports *asynchronous blocks*, *threads*, and *interrupt service routines*:

```
Async  ::= await async [ `(´LIST(Var)`)´ ] do
               Block
           end

Thread ::= await async/thread [ `(´LIST(Var)`)´ ] do
               Block
           end

Isr ::= spawn async/isr `[´ LIST(Exp) `]´ [ `(´ LIST(Var) `)´ ] do
            Block
        end

Atomic ::= atomic do
               Block
           end
```

Asynchronous execution supports tight loops while keeping the rest of the application, aka the *synchronous side*, reactive to incoming events. However, it does not support any synchronous control statement (e.g., parallel compositions, event handling, pausing, etc.).

By default, asynchronous bodies do not share variables with their enclosing scope, but the optional list of variables makes them visible to the block.

Even though asynchronous blocks execute in separate, they are still managed by the program hierarchy and are also subject to lexical scope and abortion.

Asynchronous Block

Asynchronous blocks, aka *asyncs*, intercalate execution with the synchronous side as follows:

1. Start/Resume whenever the synchronous side is idle. When multiple *asyncs* are active, they execute in lexical order.
2. Suspend after each `loop` iteration.
3. Suspend on every input `emit` (see Simulation).
4. Execute atomically and to completion unless rules `2` and `3` apply.

This rules imply that *asyncs* never execute with real parallelism with the synchronous side, preserving determinism in the program.

Examples:

```
// calculates the factorial of some "v" if it doesn't take too long
var u64  v   = <...>;
var u64  fat = 1;
var bool ok  = false;
watching 1s do
    await async (v,fat) do      // keeps "v" and "fat" visible
        loop i in [1 -> v] do   // reads from "v"
            fat = fat * i;      // writes to "fat"
        end
    end
    ok = true;                  // completed within "1s"
end
```

Simulation

An `async` block can emit input and timer events towards the synchronous side, providing a way to test programs in the language itself. Every time an `async` emits an event, it suspends until the synchronous side reacts to the event (see `rule 1` above).

Examples:

```
input int A;
```

```
// tests a program with input simulation in parallel
par do

    // original program
    var int v = await A;
    loop i in [0 -> _[ do
        await 10ms;
        _printf("v = %d\n", v+i);
    end

with

    // input simulation
    async do
        emit A(0);      // initial value for "v"
        emit 1s35ms;    // the loop in the original program executes 103 times
    end
    escape 0;

end

// The example prints the message `v = <v+i>` exactly 103 times.
```

Thread

Threads provide real parallelism for applications in Céu. Once started, a thread executes completely detached from the synchronous side. For this reason, thread execution is non deterministic and require explicit atomic blocks on accesses to variables to avoid race conditions.

A thread evaluates to a boolean value which indicates whether it started successfully or not. The value can be captured with an optional assignment.

Examples:

```
// calculates the factorial of some "v" if it doesn't take too long
var u64  v   = <...>;
var u64  fat = 1;
var bool ok  = false;
watching 1s do
    await async/thread (v,fat) do   // keeps "v" and "fat" visible
        loop i in [1 -> v] do       // reads from "v"
            fat = fat * i;          // writes to "fat"
        end
    end
    ok = true;                      // completed within "1s"
end
```

Asynchronous Interrupt Service Routine

TODO

Atomic Block

Atomic blocks provide mutual exclusion among threads, interrupts, and the synchronous side of application. Once an atomic block starts to execute, no other atomic block in the program starts.

Examples:

```
// A "race" between two threads: one incrementing, the other decrementing "count".

var s64 count = 0;                              // "count" is a shared variable
par do
    every 1s do
        atomic do
            _printf("count = %d\n", count);     // prints current value of "count" every "1s
        end
    end
with
    await async/thread (count) do
        loop do
            atomic do
                count = count - 1;              // decrements "count" as fast as possible
            end
        end
    end
with
    await async/thread (count) do
        loop do
            atomic do
                count = count + 1;              // increments "count" as fast as possible
            end
        end
    end
end
```

C Integration

Céu provides native declarations to import C symbols, native blocks to define new code in C, native statements to inline C statements, native calls to call C functions, and finalization to deal with C pointers safely:

```
Nat_Symbol ::= native [`/´(pure|const|nohold|plain)] `(´ List_Nat `)´
Nat_Block  ::= native `/´(pre|pos) do
```

```
                  <code definitions in C>
              end
Nat_End    ::= native `/´ end

Nat_Stmts  ::= `{´ {<code in C> | `@´ (`(´Exp`)´|Exp)} `}´     /* `@@´ escapes to `@´ */

Nat_Call   ::= [call] (Loc | `(´ Exp `)´)  `(´ [ LIST(Exp)] `)´

List_Nat ::= LIST(ID_nat)

Finalization ::= do [Stmt] Finalize
              |  var `&?´ Type ID_int `=´ `&´ (Call_Nat | Call_Code) Finalize
Finalize ::= finalize `(´ LIST(Loc) `)´ with
                 Block
             [ pause  with Block ]
             [ resume with Block ]
             end
```

Native calls and statements transfer execution to C, losing the guarantees of the synchronous model. For this reason, programs should only resort to C for asynchronous functionality (e.g., non-blocking I/O) or simple `struct` accessors, but never for control purposes.

`TODO: Nat_End`

Native Declaration

In Céu, any identifier prefixed with an underscore is a native symbol defined externally in C. However, all external symbols must be declared before their first use in a program.

Native declarations support four modifiers as follows:

- `const`: declares the listed symbols as constants. Constants can be used as bounded limits in vectors, pools, and numeric loops. Also, constants cannot be assigned.
- `plain`: declares the listed symbols as *plain* types, i.e., types (or composite types) that do not contain pointers. A value of a plain type passed as argument to a function does not require finalization.
- `nohold`: declares the listed symbols as *non-holding* functions, i.e., functions that do not retain received pointers after returning. Pointers passed to non-holding functions do not require finalization.
- `pure`: declares the listed symbols as pure functions. In addition to the `nohold` properties, pure functions never allocate resources that require finalization and have no side effects to take into account for the safety checks.

Examples:

```
// values
native/const  _LOW, _HIGH;      // Arduino "LOW" and "HIGH" are constants
native        _errno;           // POSIX "errno" is a global variable

// types
native/plain  _char;            // "char" is a "plain" type
native        _SDL_PixelFormat; // SDL "SDL_PixelFormat" is a type holding a pointer

// functions
native        _uv_read_start;   // Libuv "uv_read_start" retains the received pointer
native/nohold _free;            // POSIX "free" receives a pointer but does not retain it
native/pure   _strlen;          // POSIX "strlen" is a "pure" function
```

Native Block

A native block allows programs to define new external symbols in C.

The contents of native blocks is copied unchanged to the output in C depending on the modifier specified:

- `pre`: code is placed before the declarations for the Céu program. Symbols defined in `pre` blocks are visible to Céu.
- `pos`: code is placed after the declarations for the Céu program. Symbols implicitly defined by the compiler of Céu are visible to `pos` blocks.

Native blocks are copied in the order they appear in the source code.

Since Céu uses the C preprocessor, hash directives `#` inside native blocks must be quoted as `##` to be considered only in the C compilation phase.

If the code in C contains the terminating `end` keyword of Céu, the `native` block should be delimited with matching comments to avoid confusing the parser:

Symbols defined in native blocks still need to be declared for use in the program.

Examples:

```
native/plain _t;
native/pre do
    typedef int t;              // definition for "t" is placed before Céu declarations
end
var _t x = 10;                  // requires "t" to be already defined

input void A;                   // declaration for "A" is placed before "pos" blocks
native _get_A_id;
native/pos do
    int get_A_id (void) {
        return CEU_INPUT_A;     // requires "A" to be already declared
    }
end
```

```
native/nohold _printf;
native/pre do
    ##include <stdio.h>         // include the relevant header for "printf"
end

native/pos do
    /******/
    char str = "This `end` confuses the parser";
    /******/
end
```

Native Statement

The contents of native statements in between `{` and `}` are inlined in the program.

Native statements support interpolation of expressions in Céu which are expanded when preceded by the symbol `@`.

Examples:

```
var int v_ceu = 10;
{
    int v_c = @v_ceu * 2;       // yields 20
}
v_ceu = { v_c + @v_ceu };       // yields 30
{
    printf("%d\n", @v_ceu);     // prints 30
}
```

Native Call

Expressions that evaluate to a native type can be called from Céu.

If a call passes or returns pointers, it may require an accompanying finalization statement.

Examples:

```
// all expressions below evaluate to a native type and can be called

_printf("Hello World!\n");

var _t f = <...>;
f();

var _s s = <...>;
s.f();
```

Resources & Finalization

A finalization statement unconditionally executes a series of statements when its associated block terminates or is aborted.

Céu tracks the interaction of native calls with pointers and requires `finalize` clauses to accompany the calls:

- If Céu **passes** a pointer to a native call, the pointer represents a **local resource** that requires finalization. Finalization executes when the block of the local resource goes out of scope.
- If Céu **receives** a pointer from a native call return, the pointer represents an **external resource** that requires finalization. Finalization executes when the block of the receiving pointer goes out of scope.

In both cases, the program does not compile without the `finalize` statement.

A `finalize` cannot contain synchronous control statements.

Examples:

```
// Local resource finalization
watching <...> do
    var _buffer_t msg;
    <...>                       // prepares msg
    do
        _send_request(&&msg);
    finalize with
        _send_cancel(&&msg);
    end
    await SEND_ACK;             // transmission is complete
end
```

In the example above, the local variable `msg` is an internal resource passed as a pointer to `_send_request`, which is an asynchronous call that transmits the buffer in the background. If the enclosing `watching` aborts before awaking from the `await SEND_ACK`, the local `msg` goes out of scope and the external transmission would hold a *dangling pointer*. The `finalize` ensures that `_send_cancel` also aborts the transmission.

```
// External resource finalization
watching <...> do
    var&? _FILE f = &_fopen(<...>) finalize with
                        _fclose(f);
                    end;
    _fwrite(<...>, f);
    await A;
    _fwrite(<...>, f);
end
```

In the example above, the call to `_fopen` returns an external file resource as a pointer. If the enclosing `watching` aborts before awaking from the `await A`, the file would remain open as a *memory leak*. The `finalize` ensures that `_fclose` closes the file properly.

To access an external resource from Céu requires an alias assignment to a variable alias. If the external call returns `NULL` and the variable is an option alias `var&?`, the alias remains unbounded. If the variable is an alias `var&`, the assigment raises a runtime error.

Note: the compiler only forces the programmer to write finalization clauses, but cannot check if they handle the resource properly.

Declaration and expression modifiers may suppress the requirement for finalization in calls:

- `nohold` modifiers or `/nohold` typecasts make passing pointers safe.
- `pure` modifiers or `/pure` typecasts make passing pointers and returning pointers safe.
- `/plain` typecasts make return values safe.

Examples:

```
// "_free" does not retain "ptr"
native/nohold _free;
_free(ptr);
// or
(_free as /nohold)(ptr);

// "_strchr" does retain "ptr" or allocates resources
native/pure _strchr;
var _char&& found = _strchr(ptr);
// or
var _char&& found = (_strchr as /pure)(ptr);

// "_f" returns a non-pointer type
var _tp v = _f() as /plain;
```

Lua Integration

Céu provides Lua states to delimit the effects of inlined Lua statements:

```
Lua_State ::= lua `[´ [Exp] `]´ do
                  Block
              end
Lua_Stmts ::= `[´ {`=´} `[´
                  { {<code in Lua> | `@´ (`(´Exp`)´|Exp)} }   /* `@@´ escapes to `@´ */
              `]´ {`=´} `]´
```

Lua statements transfer execution to Lua, losing the guarantees of the synchronous model. For this reason, programs should only resort to C for asynchronous functionality (e.g., non-blocking I/O) or simple `struct` accessors, but never for control purposes.

All programs have an implicit enclosing *global Lua state* which all orphan statements apply.

Lua State

A Lua state creates an isolated state for inlined Lua statements.

Example:

```
// "v" is not shared between the two statements
par do
    // global Lua state
    [[ v = 0 ]];
    var int v = 0;
    every 1s do
        [[print('Lua 1', v, @v) ]];
        v = v + 1;
        [[ v = v + 1 ]];
    end
with
    // local Lua state
    lua[] do
        [[ v = 0 ]];
        var int v = 0;
        every 1s do
            [[print('Lua 2', v, @v) ]];
            v = v + 1;
            [[ v = v + 1 ]];
        end
    end
end
```

`TODO: dynamic scope, assignment/error, [dim]`

Lua Statement

The contents of Lua statements in between `[[` and `]]` are inlined in the program.

Like native statements, Lua statements support interpolation of expressions in Céu which are expanded when preceded by a `@`.

Lua statements only affect the Lua state in which they are embedded.

If a Lua statement is used in an assignment, it is evaluated as an expression that either satisfies the destination or generates a runtime error. The list that follows specifies the *Céu destination* and expected *Lua source*:

- a `var bool` expects a `boolean`
- a numeric `var` expects a `number`
- a pointer `var` expects a `lightuserdata`
- a `vector byte` expects a `string`

`TODO: lua state captures errors`

Examples:

```
var int v_ceu = 10;
[[
    v_lua = @v_ceu * 2          -- yields 20
]]
v_ceu = [[ v_lua + @v_ceu ]];   // yields 30
[[
    print(@v_ceu)               -- prints 30
]]
```

Abstractions

Céu supports reuse with `data` declarations to define new types, and `code` declarations to define new subprograms.

Declarations are subject to lexical scope.

Data

A `data` declaration creates a new data type:

```
Data ::= data ID_abs [as (nothing|Exp)] [ with
             (Var|Vec|Pool|Int) `;´ {`;´}
             { (Var|Vec|Pool|Int) `;´ {`;´} }
         end

Data_Cons ::= (val|new) Abs_Cons
Abs_Cons  ::= [Loc `.´] ID_abs `(´ LIST(Data_Cons|Vec_Cons|Exp|`nil´|`_´) `)´
```

A declaration may pack fields with storage declarations which become publicly accessible in the new type. Field declarations may assign default values for uninitialized instances.

Data types can form hierarchies using dots (`.`) in identifiers:

- An isolated identifier such as `A` makes `A` a base type.
- A dotted identifier such as `A.B` makes `A.B` a subtype of its supertype `A`.

A subtype inherits all fields from its supertype.

The optional modifier `as` expects the keyword `nothing` or a constant expression of type `int`:

- `nothing`: the `data` cannot be instantiated.
- *constant expression*: typecasting a value of the type to `int` evaluates to the specified enumeration expression.

Examples:

```
data Rect with
    var int x, y, h, w;
    var int z = 0;
end
var Rect r = val Rect(10,10, 100,100, _);  // "r.z" defaults to 0
```

```
data Dir       as nothing;  // "Dir" is a base type and cannot be intantiated
data Dir.Right as  1;       // "Dir.Right" is a subtype of "Dir"
data Dir.Left  as -1;       // "Dir.Left"  is a subtype of "Dir"
var  Dir dir = <...>;       // receives one of "Dir.Right" or "Dir.Left"
escape (dir as int);        // returns 1 or -1
```

```
TODO: new, pool, recursive types
```

Data Constructor

A new static value constructor is created in the contexts as follows:

- Prefixed by the keyword `val` in an assignment to a variable.
- As an argument to a `code` invocation.
- Nested as an argument in a `data` creation (i.e., a `data` that contains another `data`).

In all cases, the arguments are copied to a destination with static storage. The destination must be a plain declaration (i.e., not an alias or pointer).

The constructor uses the `data` identifier followed by a list of arguments matching the fields of the type.

Variables of the exact same type can be copied in assignments.

For assignments from a subtype to a supertype, the rules are as follows:

- Copy assignments
  - plain values: only if the subtype contains no extra fields
  - pointers: allowed
- Alias assignment: allowed.

```
data Object with
    var Rect rect;
    var Dir  dir;
```

```
end
var Object o1 = val Object(Rect(0,0,10,10,_), Dir.Right());
var Object o2 = o1;         // makes a deep copy of all fields from "o1" to "o2"
```

Code

The `code/tight` and `code/await` declarations specify new subprograms that can be invoked from arbitrary points in programs:

```
// prototype declaration
Code_Tight ::= code/tight Mods ID_abs `(´ Params `)´ `->´ Type
Code_Await ::= code/await Mods ID_abs `(´ Params `)´ [`->´ `(´ Params `)´] `->´ (Type | FORE
Params ::= void | LIST(Var|Vec|Pool|Int)

// full declaration
Code_Impl ::= (Code_Tight | Code_Await) do
                  Block
              end

// invocation
Code_Call  ::= call  Mods Abs_Cons
Code_Await ::= await Mods Abs_Cons
Code_Spawn ::= spawn Mods Abs_Cons [in Loc]

Mods ::= [`/´dynamic | `/´static] [`/´recursive]
```

A `code/tight` is a subprogram that cannot contain synchronous control statements and runs to completion in the current internal reaction.

A `code/await` is a subprogram with no restrictions (e.g., it can manipulate events and use parallel compositions) and its execution may outlive multiple reactions.

A *prototype declaration* specifies the interface parameters of the abstraction which invocations must satisfy. A *full declaration* (aka *definition*) also specifies an implementation with a block of code. An *invocation* specifies the name of the code abstraction and arguments matching its declaration.

To support recursive abstractions, a code invocation can appear before the implementation is known, but after the prototype declaration. In this case, the declaration must use the modifier `/recursive`.

Examples:

```
code/tight Absolute (var int v) -> int do   // declares the prototype for "Absolute"
    if v > 0 then                           // implements the behavior
        escape  v;
    else
```

```
        escape -v;
    end
end
var int abs = call Absolute(-10);           // invokes "Absolute" (yields 10)

code/await Hello_World (void) -> FOREVER do
    every 1s do
        _printf("Hello World!\n");  // prints "Hello World!" every second
    end
end
await Hello_World();                // never awakes

code/tight/recursive Fat (var int v) -> int;    // "Fat" is a recursive code
code/tight/recursive Fat (var int v) -> int do
    if v > 1 then
        escape v * (call/recursive Fat(v-1));   // recursive invocation before full declarat
    else
        escape 1;
    end
end
var int fat = call/recursive Fat(10);           // invokes "Fat" (yields 3628800)

TODO: hold
```

Code Declaration

Code abstractions specify a list of input parameters in between the symbols `(` and `)`. Each parameter specifies an entity class with modifiers, a type and an identifier. A `void` list specifies that the abstraction has no parameters.

Code abstractions also specify an output return type. A `code/await` may use `FOREVER` as output to indicate that it never returns.

A `code/await` may also specify an optional *public parameter list*, which are local storage entities living the outermost scope of the abstraction body. These entities are visible to the invoking context which may access them while the abstraction executes.

Code Invocation

A `code/tight` is invoked with the keyword `call` followed by the abstraction name and list of arguments. A `code/await` is invoked with the keywords `await` or `spawn` followed by the abstraction name and list of arguments.

The list of arguments must satisfy the list of parameters in the code declaration.

The `call` and `await` invocations suspend the current trail and transfer control to the code abstraction. The invoking point only resumes after the abstraction terminates and evaluates to a value of its return type which can be captured with an optional assignment.

The `spawn` invocation also suspends and transfers control to the code abstraction. However, when the abstraction becomes idle (or terminates), the invoking point resumes. This makes the invocation point and a non-terminating abstraction to execute concurrently.

The `spawn` invocation also accepts an optional pool which provides storage and scope for invoked abstractions.

If the `spawn` provides the pool, the invocation evaluates to a boolean that indicates whether the pool has space to execute the code. The result can be captured with an optional assignment. If the pool goes out of scope, all invoked abstractions residing in that pool are aborted.

If the `spawn` omits the pool, the invocation always succeed and has the same scope as the invoking point: when the enclosing block terminates, the invoked code is also aborted.

Dynamic Dispatching

Céu supports dynamic code dispatching based on multiple parameters.

The modifier `/dynamic` in a declaration specifies that the code is dynamically dispatched. A dynamic code must have at least one `dynamic` parameter. Also, all dynamic parameters must be pointers or aliases to a data type in some hierarchy.

A dynamic declaration requires other compatible dynamic declarations with the same name, modifiers, parameters, and return type. The exceptions are the `dynamic` parameters, which must be in the same hierarchy of their corresponding parameters in other declarations.

To determine which declaration to execute during runtime, the actual argument runtime type is checked against the first formal `dynamic` parameter of each declaration. The declaration with the most specific type matching the argument wins. In the case of a tie, the next dynamic parameter is checked.

A *catchall* declaration with the most general dynamic types must always be provided.

If the argument is explicitly typecast to a supertype, then dispatching considers that type instead.

Example:

```
data Media as nothing;
data Media.Audio     with <...> end
data Media.Video     with <...> end
data Media.Video.Avi with <...> end
```

```
code/await/dynamic Play (dynamic var& Media media) -> void do
    _assert(0);             // never dispatched
end
code/await/dynamic Play (dynamic var& Media.Audio media) -> void do
    <...>                   // plays an audio
end
code/await/dynamic Play (dynamic var& Media.Video media) -> void do
    <...>                   // plays a video
end
code/await/dynamic Play (dynamic var& Media.Video.Avi media) -> void do
    <...>                                   // prepare the avi video
    await/dynamic Play(&m as Media.Video);  // dispatches the supertype
end

var& Media m = <...>;       // receives one of "Media.Audio" or "Media.Video"
await/dynamic Play(&m);     // dispatches the appropriate subprogram to play the media
```

Synchronous Control Statements

The *synchronous control statements* which follow cannot appear in event iterators, pool iterators, asynchronous execution, finalization, and tight code abstractions: `await`, `spawn`, `emit` (internal events), `every`, `finalize`, `pause/if`, `par`, `par/and`, `par/or`, and `watching`.

As exceptions, an `every` can `emit` internal events, and a `code/tight` can contain empty `finalize` statements.

Locations & Expressions

Locations & Expressions

Céu specifies locations and expressions as follows:

```
Exp ::= NUM | STR | null | true | false
     |  `(´ Exp `)´
     |  Exp <binop> Exp
     |  <unop> Exp
     |  Exp `[´ Exp `]´
     |  Exp is Type
     |  Exp as Type
     |  Exp as `/´(nohold|plain|pure)
     |  sizeof `(´ (Type|Exp) `)´
     |  Nat_Call | Code_Call
```

```
/* Locations */

Loc ::= Loc [as (Type | `/´(nohold|plain|pure)) `)´
     |  [`*´|`$´] Loc
     |  Loc { `[´Exp`]´ | (`:´|`.´) (ID_int|ID_nat) | `!´ }
     |  ID_int
     |  ID_nat
     |  outer
     |  `{´ <code in C> `}´
     |  `(´ Loc `)´

/* Operator Precedence */

    /* lowest priority */

    // locations
    *     $
    :     .     !     []
    as

    // expressions
    is    as                                         // binops
    or
    and
    !=    ==    <=    >=    <     >
    |
    ^
    &
    <<    >>
    +     -
    *     /     %
    not   +     -     ~     $$    $     *     &&    &   // unops
    :     .     !     ?     ()    []

    /* highest priority */
```

Primary

`TODO`

Arithmetic

Céu supports the arithmetic expressions *addition*, *subtraction*, *modulo (remainder)*, *multiplication*, *division*, *unary-plus*, and *unary-minus* through the operators that follow:

```
+      -      %      *      /      +      -
```

Bitwise

Céu supports the bitwise expressions *not*, *and*, *or*, *xor*, *left-shift*, and *right-shift* through the operators that follow:

```
~      &      |      ^      <<      >>
```

Relational

Céu supports the relational expressions *equal-to*, *not-equal-to*, *greater-than*, *less-than*, *greater-than-or-equal-to*, and *less-than-or-equal-to* through the operators that follow:

```
==      !=      >      <      >=      <=
```

Relational expressions evaluate to *true* or *false*.

Logical

Céu supports the logical expressions *not*, *and*, and *or* through the operators that follow:

```
not      and      or
```

Logical expressions evaluate to *true* or *false*.

Types

Céu supports type checks and casts:

```
Check ::= Exp is Type
Cast  ::= Exp as Type
```

Type Check

A type check evaluates to *true* or *false* depending on whether the runtime type of the expression is a subtype of the checked type or not.

The static type of the expression must be a supertype of the checked type.

Example:

```
data Aa;
data Aa.Bb;
var Aa a = <...>;        // "a" is of static type "Aa"
<...>
if a is Aa.Bb then       // is the runtime type of "a" a subtype of "Aa.Bb"?
    <...>
end
```

Type Cast

A type cast converts the type of an expression into a new type as follows:

1. The expression type is a data type:
    1. The new type is `int`: Evaluates to the type enumeration for the expression type.
    2. The new type is a subtype of the expression static type:
        1. The expression runtime type is a subtype of the new type: Evaluates to the new type.
        2. Evaluates to error.
    3. The new type is a supertype of the expression static type: Always succeeds and evaluates to the new type. See also Dynamic Dispatching.
    4. Evaluates to error.
2. Evaluates to the new type (i.e., a *weak typecast*, as in C).

Examples:

```
var Direction dir = <...>;
_printf("dir = %d\n", dir as int);

var Aa a = <...>;
_printf("a.v = %d\n", (a as Aa.Bb).v);

var Media.Video vid = <...>;
await/dynamic Play(&m as Media);

var bool b = <...>;
_printf("b= %d\n", b as int);
```

Modifiers

Expressions that evaluate to native types can be modified as follows:

```
Mod ::= Exp as `/´(nohold|plain|pure)
```

Modifiers may suppress the requirement for resource finalization.

References

Céu supports *aliases* and *pointers* as references.

Aliases

An alias is acquired by prefixing a native call or a location with the operator `&`:

```
Alias ::= `&´ (Nat_Call | Loc)
```

See also the unwrap operator `!` for option variable aliases.

Pointers

The operator `&&` returns the address of a location, while the operator `*` dereferences a pointer:

```
Addr  ::= `&&´ Loc
Deref ::= `*´ Loc
```

Option

The operator `?` checks if the location of an option type is set, while the operator `!` unwraps the location, raising an error if it is unset:

```
Check  ::= Loc `?´
Unwrap ::= Loc `!´
```

Sizeof

A `sizeof` expression returns the size of a type or expression, in bytes:

```
Sizeof ::= sizeof `(´ (Type|Exp) `)´
```

Calls

See Native Call and Code Invocation.

Vectors

Index

Céu uses square brackets to index vectors:

```
Vec_Idx ::= Loc `[´ Exp `]´
```

The index expression must be of type `usize`.

Vectors start at index zero. Céu generates an error for out-of-bounds vector accesses.

Length

The operator `$` returns the current length of a vector, while the operator `$$` returns the max length:

```
Vec_Len ::= `$´  Loc
Vec_Max ::= `$$´ Loc
```

`TODO: max`

The vector length can also be assigned:

```
vector[] int vec = [ 1, 2, 3 ];
$vec = 1;
```

The new length must be smaller or equal to the current length, otherwise the assignment raises a runtime error. The space for dynamic vectors shrinks automatically.

Constructor

Vector constructors are only valid in assignments:

```
Vec_Cons   ::= (Loc | Exp) Vec_Concat { Vec_Concat }
            |  `[´ [LIST(Exp)] `]´ { Vec_Concat }
Vec_Concat ::= `..´ (Exp | Lua_Stmts | `[´ [LIST(Exp)] `]´)
```

Examples:

```
vector[3] int v;     // declare an empty vector of length 3     (v = [])
v = v .. [8];        // append value '8' to the empty vector    (v = [8])
v = v .. [1] .. [5]; // append values '1' and '5' to the vector (v = [8, 1, 5])
```

Fields

The operators `.´ and`:´ specify fields of data abstractions and native structs:

```
Dot   ::= Loc `.´ (ID_int|ID_nat)
Colon ::= Loc `:´ (ID_int|ID_nat)
```

The expression `e:f` is a sugar for `(*e).f`.

`TODO: ID_nat to avoid clashing with Céu keywords.`

Compilation

Compilation

The compiler converts an input program in Céu to an output in C, which is further embedded in an environment satisfying a C API, which is finally compiled to an executable:

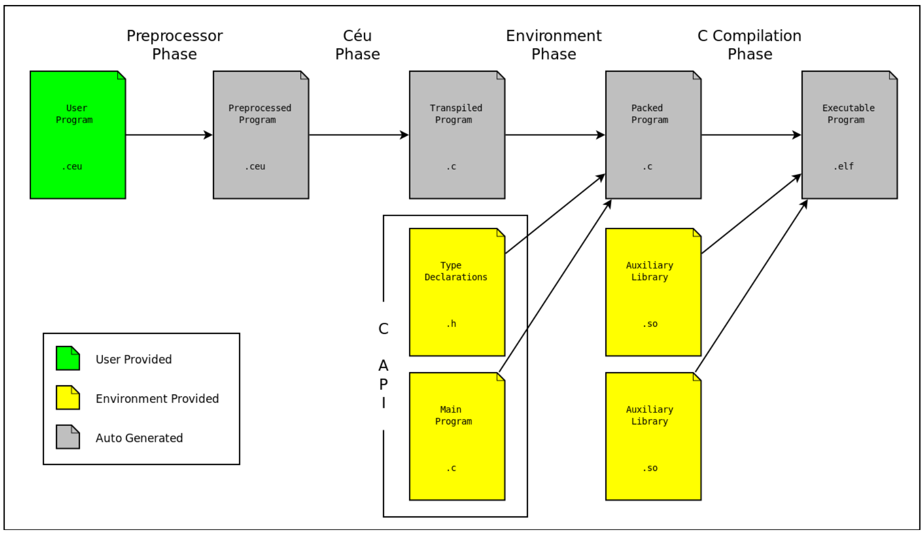

Figure 3:

Command Line

The single command `ceu` is used for all compilation phases:

```
Usage: ceu [<options>] <file>...

Options:

    --help                      display this help, then exit
    --version                   display version information, then exit

    --pre                       Preprocessor Phase: preprocess Céu into Céu
    --pre-exe=FILE                  preprocessor executable
    --pre-args=ARGS                 preprocessor arguments
    --pre-input=FILE                input file to compile (Céu source)
    --pre-output=FILE               output file to generate (Céu source)
```

```
--ceu                         Céu Phase: compiles Céu into C
--ceu-input=FILE                  input file to compile (Céu source)
--ceu-output=FILE                 output source file to generate (C source)
--ceu-line-directives=BOOL        insert `#line´ directives in the C output

--ceu-features-lua=BOOL           enable `lua´ support
--ceu-features-thread=BOOL        enable `async/thread´ support
--ceu-features-isr=BOOL           enable `async/isr´ support

--ceu-err-unused=OPT              effect for unused identifier: error|warning|pass
--ceu-err-unused-native=OPT                  unused native identifier
--ceu-err-unused-code=OPT                    unused code identifier
--ceu-err-uninitialized=OPT       effect for uninitialized variable: error|warning|pass

--env                         Environment Phase: packs all C files together
--env-types=FILE                  header file with type declarations (C source)
--env-threads=FILE                header file with thread declarations (C source)
--env-ceu=FILE                    output file from Céu phase (C source)
--env-main=FILE                   source file with main function (C source)
--env-output=FILE                 output file to generate (C source)

--cc                          C Compiler Phase: compiles C into binary
--cc-exe=FILE                     C compiler executable
--cc-args=ARGS                    compiler arguments
--cc-input=FILE                   input file to compile (C source)
--cc-output=FILE                  output file to generate (binary)
```

All phases are optional. To enable a phase, the associated prefix must be enabled. If two consecutive phases are enabled, the output of the preceding and the input of the succeeding phases can be omitted.

Examples:

```
## Preprocess "user.ceu", and converts the output to "user.c"
$ ceu --pre --pre-input="user.ceu" --ceu --ceu-output="user.c"
```

```
## Packs "user.c", "types.h", and "main.c", compiling them to "app.out"
$ ceu --env --env-ceu=user.c --env-types=types.h --env-main=main.c \
      --cc --cc-output=app.out
```

C API

The environment phase of the compiler packs the converted Céu program and additional files in the order as follows:

1. type declarations (option `--env-types`)
2. thread declarations (option `--env-threads`, optional)

3. a callback prototype (fixed, see below)
4. Céu program (option `--env-ceu`, auto generated)
5. main program (option `--env-main`)

The Céu program uses standardized types and calls, which must be previously mapped from the host environment in steps `1-3`.

The main program depends on declarations from the Céu program.

Types

The type declarations must map the types of the host environment to all primitive types of Céu.

Example:

```
##include <stdint.h>
##include <sys/types.h>

typedef unsigned char bool;
typedef unsigned char byte;
typedef unsigned int  uint;

typedef ssize_t  ssize;
typedef size_t   usize;

typedef int8_t    s8;
typedef int16_t  s16;
typedef int32_t  s32;
typedef int64_t  s64;

typedef uint8_t   u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;

typedef float    f32;
typedef double   f64;
```

Threads

If the user program uses threads and the option `--ceu-features-thread` is set, the host environment must provide declarations for types and functions expected by Céu.

Example:

```
##include <pthread.h>
##include <unistd.h>
##define CEU_THREADS_T                pthread_t
##define CEU_THREADS_MUTEX_T          pthread_mutex_t
##define CEU_THREADS_CREATE(t,f,p)    pthread_create(t,NULL,f,p)
##define CEU_THREADS_CANCEL(t)        ceu_dbg_assert(pthread_cancel(t)==0)
##define CEU_THREADS_JOIN_TRY(t)      0
##define CEU_THREADS_JOIN(t)          ceu_dbg_assert(pthread_join(t,NULL)==0)
##define CEU_THREADS_MUTEX_LOCK(m)    ceu_dbg_assert(pthread_mutex_lock(m)==0)
##define CEU_THREADS_MUTEX_UNLOCK(m) ceu_dbg_assert(pthread_mutex_unlock(m)==0)
##define CEU_THREADS_SLEEP(us)        usleep(us)
##define CEU_THREADS_PROTOTYPE(f,p)   void* f (p)
##define CEU_THREADS_RETURN(v)        return v
```

```
TODO: describe them
```

Céu

The converted program generates types and constants required by the main program.

External Events

For each external input and output event `<ID>` defined in Céu, the compiler generates corresponding declarations as follows:

1. An enumeration item `CEU_INPUT_<ID>` that univocally identifies the event.
2. A `define` macro `_CEU_INPUT_<ID>_`.
3. A struct type `tceu_input_<ID>` with fields corresponding to the types in of the event payload.

Example:

Céu program:

```
input (int,u8&&) MY_EVT;
```

Converted program:

```
enum {
    ...
    CEU_INPUT_MY_EVT,
    ...
};

##define _CEU_INPUT_MY_EVT_

typedef struct tceu_input_MY_EVT {
    int _1;
```

```
    u8* _2;
} tceu_input_MY_EVT;
```

Data

The global `CEU_APP` of type `tceu_app` holds all program memory and runtime information:

```
typedef struct tceu_app {
    bool end_ok;                /* if the program terminated */
    int  end_val;               /* final value of the program */
    bool async_pending;         /* if there is a pending "async" to execute */
    ...
    tceu_code_mem_ROOT root;    /* all Céu program memory */
} tceu_app;

static tceu_app CEU_APP;
```

The struct `tceu_code_mem_ROOT` holds the whole memory of the Céu program. The identifiers for global variables are preserved, making them directly accessible.

Example:

```
var int x = 10;
```

```
typedef struct tceu_code_mem_ROOT {
    ...
    int  x;
} tceu_code_mem_ROOT;
```

Main

The main program provides the entry point for the host platform (i.e., the `main` function), implementing the event loop that senses the world and notifies the Céu program about changes.

The main program interfaces with the Céu program in both directions:

- Through direct calls, in the direction `main -> Céu`, typically when new input is available.
- Through callbacks, in the direction `Céu -> main`, typically when new output is available.

Calls

The functions that follow are called by the main program to command the execution of Céu programs:

- `void ceu_start (tceu_callback* cb, int argc, char* argv[])`

  Initializes and starts the program. Should be called once. Expects a callback to register for further notifications. Also receives the program arguments in `argc` and `argv`.

- `void ceu_stop  (void)`

  Finalizes the program. Should be called once.

- `void ceu_input (tceu_nevt evt_id, void* evt_params)`

  Notifies the program about an input `evt_id` with a payload `evt_params`. Should be called whenever the event loop senses a change. The call to `ceu_input(CEU_INPUT__ASYNC, NULL)` makes asynchronous blocks to execute a step.

- `int ceu_loop (tceu_callback* cb, int argc, char* argv[])`

  Implements a simple loop encapsulating `ceu_start`, `ceu_input`, and `ceu_stop`. On each loop iteration, make a `CEU_CALLBACK_STEP` callback and generates a `CEU_INPUT__ASYNC` input. Should be called once. Returns the final value of the program.

- `void ceu_callback_register (tceu_callback* cb)`

  Registers a new callback.

Callbacks

The Céu program makes callbacks to the main program in specific situations:

```
enum {
    CEU_CALLBACK_START,                 /* once in the beginning of `ceu_start`
    CEU_CALLBACK_STOP,                  /* once in the end of `ceu_stop`
    CEU_CALLBACK_STEP,                  /* on every iteration of `ceu_loop`
    CEU_CALLBACK_ABORT,                 /* whenever an error occurs
    CEU_CALLBACK_LOG,                   /* on error and debugging messages
    CEU_CALLBACK_TERMINATING,           /* once after executing the last statement
    CEU_CALLBACK_ASYNC_PENDING,         /* whenever there's a pending "async" block
    CEU_CALLBACK_THREAD_TERMINATING,    /* whenever a thread terminates
    CEU_CALLBACK_ISR_ENABLE,            /* whenever interrupts should be enabled/disabled
    CEU_CALLBACK_ISR_ATTACH,            /* whenever an "async/isr" starts
    CEU_CALLBACK_ISR_DETACH,            /* whenever an "async/isr" is aborted
    CEU_CALLBACK_ISR_EMIT,              /* whenever an "async/isr" emits an innput
    CEU_CALLBACK_WCLOCK_MIN,            /* whenever a next minimum timer is required
    CEU_CALLBACK_WCLOCK_DT,             /* whenever the elapsed time is requested
    CEU_CALLBACK_OUTPUT,                /* whenever an output is emitted
    CEU_CALLBACK_REALLOC,               /* whenever memory is allocated/deallocated
};
```

TODO: payloads

Céu invokes the registered callbacks in reverse register order, one after the other, stopping when a callback returns that it handled the request.

A callback is composed of a function handler and a pointer to the next callback:

```
typedef struct tceu_callback {
    tceu_callback_f       f;
    struct tceu_callback* nxt;
} tceu_callback;
```

The handler should expect a request identifier with two arguments, as well as the filename and line number in the source code in Céu making the request:

```
typedef tceu_callback_ret (*tceu_callback_f) (int, tceu_callback_arg, tceu_callback_arg, co
```

An argument has one of the following types:

```
typedef union tceu_callback_arg {
    void* ptr;
    s32   num;
    usize size;
} tceu_callback_arg;
```

The handler returns if it handled the request and an optional value:

```
typedef struct tceu_callback_ret {
    bool is_handled;
    tceu_callback_arg value;
} tceu_callback_ret;
```

Example

Suppose the environment supports the events that follow:

```
input  int I;
output int O;
```

The `main.c` implements an event loop to sense occurrences of `I` and a callback to handle occurrences of `O`:

```
##include "types.h"      // as illustrated above in "Types"

int ceu_is_running;     // detects program termination

tceu_callback_ret ceu_callback_main (int cmd, tceu_callback_arg p1, tceu_callback_arg p2, c
{
    tceu_callback_ret ret = { .is_handled=0 };
    switch (cmd) {
        case CEU_CALLBACK_TERMINATING:
```

```
            ceu_is_running = 0;
            ret.is_handled = 1;
            break;
        case CEU_CALLBACK_OUTPUT:
            if (p1.num == CEU_OUTPUT_O) {
                printf("output O has been emitted with %d\n", p2.num);
                ret.is_handled = 1;
            }
            break;
    }
    return ret;
}

int main (int argc, char* argv[])
{
    ceu_is_running = 1;
    tceu_callback cb = { &ceu_callback_main, NULL };
    ceu_start(&cb, argc, argv);

    while (ceu_is_running) {
        if (<call-to-detect-if-A-occurred>()) {
            int v = <argument-to-A>;
            ceu_input(CEU_INPUT_A, &v);
        }
        ceu_input(CEU_INPUT__ASYNC, NULL);
    }

    ceu_stop();
}
```

Syntax

Syntax

Follows the complete syntax of Céu in a BNF-like syntax:

- `A` : non terminal (starting in uppercase)
- **`a`** : terminal (in bold and lowercase)
- ‘.´ : terminal (non-alphanumeric characters)
- `A ::= ...` : defines `A` as `...`
- `x y` : `x` in sequence with `y`
- `x|y` : `x` or `y`
- `{x}` : zero or more xs
- `[x]` : optional x

- `LIST(x)` : expands to x {‘,´ x} [‘,´]
- `(...)` : groups ...
- `<...>` : special informal rule

```
Program ::= Block
Block   ::= {Stmt `;´} {`;´}

Stmt ::= nothing

  /* Blocks */

      // Do ::=
      | do [`/´(ID_int|`_´)] [`(´ [LIST(ID_int)] `)´]
            Block
        end
      |  escape [`/´ID_int] [Exp]

      /* pre (top level) execution */
      | pre do
            Block
        end

  /* Storage Entities / Declarations */

      // Dcls ::=
      | var [`&´|`&?´] [`/dynamic´|`/nohold´] Type ID_int [`=´ Sources]
      | vector [`&´] `[´ [Exp] `]´ Type ID_int [`=´ Sources]
      | pool [`&´] `[´ [Exp] `]´ Type ID_int [`=´ Sources]
      | event [`&´] (Type | `(´ LIST(Type) `)´) ID_int [`=´ Sources]

      | input (Type | `(´ LIST(Type) `)´) ID_ext
      | output (Type | `(´ LIST([`&´] Type) `)´) ID_ext

  /* Event Handling */

      // Await ::=
      | await (ID_ext | Loc) [until Exp]
      | await (WCLOCKK|WCLOCKE)
      //
      | await (FOREVER | pause | resume)

      // Emit_Ext ::=
      | emit ID_ext [`(´ [LIST(Exp)] `)´]
      | emit (WCLOCKK|WCLOCKE)
      //
      | emit Loc [`(´ [LIST(Exp)] `)´]
```

```
    | lock Loc do
          Block
      end

/* Conditional */

    | if Exp then
          Block
      { else/if Exp then
          Block }
      [ else
          Block ]
      end

/* Loops */

    /* simple */
    | loop [`/´Exp] do
          Block
      end

    /* numeric iterator */
    | loop [`/´Exp] (ID_int|`_´) in [Range] do
          Block
      end
      // where
          Range ::= (`[´ | `]´)
                      ( (      Exp `->´ (Exp|`_´))
                      | ((Exp|`_´) `<-´ Exp      ) )
                    (`[´ | `]´) [`,´ Exp]

    /* pool iterator */
    | loop [`/´Exp] (ID_int|`_´) in Loc do
          Block
      end

    /* event iterator */
    | every [(Loc | `(´ LIST(Loc|`_´) `)´) in] (ID_ext|Loc|WCLOCKK|WCLOCKE) do
          Block
      end

    |  break [`/´ID_int]
    |  continue [`/´ID_int]

/* Parallel Compositions */
```

```
    /* parallels */
    | (par | par/and | par/or) do
          Block
      with
          Block
      { with
          Block }
       end

    /* watching */
    // Watching ::=
    | watching LIST(ID_ext|Loc|WCLOCKK|WCLOCKE|Code_Cons_Init) do
          Block
      end

    /* block spawn */
    | spawn [`(´ [LIST(ID_int)] `)´] do
          Block
      end

/* Pause */

    | pause/if (Loc|ID_ext) do
          Block
      end

/* Asynchronous Execution */

    | await async [ `(´ LIST(Var) `)´ ] do
          Block
      end

    // Thread ::=
    | await async/thread [ `(´ LIST(Var) `)´ ] do
          Block
      end

    | spawn async/isr `[´ LIST(Exp) `]´ [ `(´ LIST(Var) `)´ ] do
          Block
      end

    /* synchronization */
    | atomic do
          Block
      end
```

```
/* C integration */

    | native [`/´(pure|const|nohold|plain)] `(´ List_Nat `)´
      // where
          List_Nat ::= LIST(ID_nat)
    | native `/´(pre|pos) do
          <code definitions in C>
      end
    | native `/´ end
    | `{´ {<code in C> | `@´ (`(´Exp`)´|Exp)} `}´     /* `@@´ escapes to `@´ */

    // Nat_Call ::=
    | [call] Exp

    /* finalization */
    | do [Stmt] Finalize
    | var `&?´ Type ID_int `=´ `&´ (Nat_Call | Code_Call) Finalize
      // where
          Finalize ::= finalize `(´ LIST(Loc) `)´ with
                           Block
                       [ pause  with Block ]
                       [ resume with Block ]
                       end

/* Lua integration */

    // Lua_State ::=
    | lua `[´ [Exp] `]´ do
          Block
      end
    // Lua_Stmts ::=
    | `[´ {`=´} `[´
          { {<code in Lua> | `@´ (`(´Exp`)´|Exp)} }   /* `@@´ escapes to `@´ */
      `]´ {`=´} `]´

/* Abstractions */

    /* Data */

    | data ID_abs [as (nothing|Exp)] [ with
          Dcls `;´ {`;´}
          { Dcls `;´ {`;´} }
      end ]

    /* Code */
```

```
    // Code_Tight ::=
    | code/tight Mods ID_abs `(´ Params `)´ `->´ Type

    // Code_Await ::=
    | code/await Mods ID_abs `(´ Params `)´ [ `->´ `(´ Params `)´ ] `->´ (Type | FOREVER)
      // where
          Params ::= void | LIST(Dcls)

    /* code implementation */
    | (Code_Tight | Code_Await) do
          Block
      end

    /* code invocation */

    // Code_Call ::=
    | call  Mods Abs_Cons

    // Code_Await ::=
    | await Mods Abs_Cons

    // Code_Spawn ::=
    | spawn Mods Abs_Cons [in Loc]

      // where
          Mods ::= [`/´dynamic | `/´static] [`/´recursive]
          Abs_Cons ::= [Loc `.´] ID_abs `(´ LIST(Data_Cons|Vec_Cons|Exp|`nil´|`_´) `)´

/* Assignments */

    | (Loc | `(´ LIST(Loc|`_´) `)´) `=´ Sources
      // where
          Sources ::= ( Do
                      | Emit_Ext
                      | Await
                      | Watching
                      | Thread
                      | Lua_State
                      | Lua_Stmts
                      | Code_Await
                      | Code_Spawn
                      | Vec_Cons
                      | Data_Cons
                      | Exp
                      | `nil´
```

```
                       | `_´ )
            Vec_Cons  ::= (Loc | Exp) Vec_Concat { Vec_Concat }
                       |  `[´ [LIST(Exp)] `]´ { Vec_Concat }
                        // where
                            Vec_Concat ::= `..´ (Exp | Lua_Stmts | `[´ [LIST(Exp)] `]´)
            Data_Cons ::= (val|new) Abs_Cons

/* Identifiers */

ID       ::= [a-zA-Z0-9_]+
ID_int   ::= ID             // ID beginning with lowercase
ID_ext   ::= ID             // ID all in uppercase, not beginning with digit
ID_abs   ::= ID {`.´ ID}    // IDs beginning with uppercase, containining at least one lower
ID_field ::= ID             // ID not beginning with digit
ID_nat   ::= ID             // ID beginning with underscore
ID_type  ::= ( ID_nat | ID_abs
             | void  | bool  | byte
             | f32   | f64   | float
             | s8    | s16   | s32   | s64
             | u8    | u16   | u32   | u64
             | int   | uint  | ssize | usize )

/* Types */

Type ::= ID_type { `&&´ } [`?´]

/* Wall-clock values */

WCLOCKK ::= [NUM h] [NUM min] [NUM s] [NUM ms] [NUM us]
WCLOCKE ::= `(´ Exp `)´ (h|min|s|ms|us)

/* Literals */

NUM ::= [0-9] ([0-9]|[xX]|[A-F]|[a-f]|\.)*  // regex
STR ::= " [^\"\n]* "                        // regex

/* Expressions */

Exp ::= NUM | STR | null | true | false
     |  `(´ Exp `)´
     |  Exp <binop> Exp
     |  <unop> Exp
     |  Exp `[´ Exp `]´
     |  Exp is Type
     |  Exp as Type
     |  Exp as `/´(nohold|plain|pure)
```

```
     |  sizeof `(´ (Type|Exp) `)´
     |  Nat_Call | Code_Call

/* Locations */

Loc ::= Loc [as (Type | `/´(nohold|plain|pure)) `)´
     |  [`*´|`$´] Loc
     |  Loc { `[´Exp`]´ | (`:´|`.´) (ID_int|ID_nat) | `!´ }
     |  ID_int
     |  ID_nat
     |  outer
     |  `{´ <code in C> `}´
     |  `(´ Loc `)´

/* Operator Precedence */

    /* lowest priority */

    // locations
    *     $
    :     .     !     []
    as

    // expressions
    is    as                                        // binops
    or
    and
    !=    ==    <=    >=    <     >
    |
    ^
    &
    <<    >>
    +     -
    *     /     %
    not   +     -     ~     $$    $     *     &&    &   // unops
    :     .     !     ?     ()    []

    /* highest priority */

/* Other */

    // single-line comment

    /** nested
        /* multi-line */
        comments **/
```

```
# preprocessor directive
```

TODO: statements that do not require ;

License

License

Céu is distributed under the MIT license reproduced below: