Dew language specification

by Yucheng Zhang 2019–09–10

1 Background

A stream is basically a sequence of values, in which each value describes the same thing at different consecutive times. To put it in another way, streams can be used to describe things which changes over time, and thus has a huge potential use in game development. For example, the physical location of an apple could be described as stream, or the state of an NPC could be described as a stream. A data-flow language sees streams as first-class citizens, and can be used to easily describe relations of many streams.

Two related areas are: (1) data-flow languages used in embedded control systems, e.g. Lucid Synchrone, (2) functional reactive programming, e.g. Elerea library written in Haskell. However, They are not exactly suitable for practical game development, probably because they were not designed specificially for game development in the first place. We hope to design a new language for practical game development. To do so, we have two major goals: (1) the language must excel in expressive power; (2) the language must be easy to use.

Expressive power with stream-based languages have two aspects: (1) dynamism within the stream network, (2) dynamism with interacting with outside world. For (1), stream networks traditional data-flow languages are mostly static, especially for those used for embedded control systems, where predictability is more important. For (2), previous works usually designate a fixed set of streams for input and output.

Elerea presented an elegant and powerful model for describing dynamic stream networks, using two monads `Signal` and `SignalGen`. For game development, Elerea has two issues: (1) interacting with the outside world is supposed to go through a fixed set of streams; (2) a typical program written with Elerea is riddled with monadic operators, and have a high cognitive burden.

We improve upon Elerea as follows:

1. We solve the IO dynamism problem by allowing each stream to have side-effects to interact with the outside world. Side-effects in streams are executed once per frame throughout the stream's lifetime, which is largely determined by GC aliveness of the stream. We develop our own compiler and runtime to provide strong guarantees about stream lifetimes. We also introduced two operators `switch` and `keepalive` for controlling stream lifetimes.

   Technically, Elerea also allows side-effects in streams, but it's probably not the intended usage. Side-effects in Elerea is also depending on GC aliveness, which is practically uncontrollable since Elerea is developed as a Haskell library.

2. We reduced programmer cognitive overhead significantly, and our language is only a few more stream primitives than a common programming language. Essentially, every expression in Dew is in the `SignalGen` monad, and `dew -> expr` is similar to the `generator` combinator in Elerea, but with the type `generator' :: SignalGen a -> SignalGen (Signal a)` instead. We still retain equivalent expressive power after the simplification: (1) for the Signal monad, because the same results can be achieved easily with `dew -> expr` and `@expr`; (2) for the original `generator`, we can emulate with `dew -> expr` and a stream of thunks.

To make the language comfortable to use as a game script language, we also have the following additional design goals:

1. The language should have static types with type inference. We don't want to manually specify types for quick scripting, but we also need static types for both performance and confidence of correctness when editing the game scripts.

2. The language should be mostly immutable, because values that are mutable over time are better expressed as streams. On the other hand, we need imperative features (1) to interact with the game core engine, and (2) for convenience when scripting.

3. We want to design the language to be an expression-based language, where most language constructs are expressions. This is to make using results from previous computations easier.

To meet the above design goals, we designed the language as a variant of ML, and with a variant of Hindley-Milner type inference.

2 Types

```
typexpr ::= ' ident
          | '' ident
          | _
          | num
          | bool
          | keyword
          | external-type
          | ()
          | typexpr {, typexpr}+
          | (| [recfield-type {; recfield-type} [;]] [| ident] |)
          | [ typexpr ]
          | [| typexpr |]
          | ? typexpr
          | typexpr {\ typexpr}+
          | typeconstr
          | typexpr typeconstr
          | ( typexpr [; typexpr] [;] ) typeconstr
          | typexpr -> typexpr
          | ( typexpr )
          | * typexpr
          | ( n *) typexpr

type-definition ::= type typedef {and typedef}
typedef ::= [typeparams] typeconstr = [|] constr {| constr}
constr ::= ident of typexpr
typeparams ::= typeparam |
             | ( typeparam {; typeparam} [;] )
typeparam ::= '' ident

recfield-type ::= ident : typexpr
```

`'ident`, and `''ident` are type variables. `''ident` are usual type variables; and `'ident` have the restriction that can only take any type that is not a stream, i.e. not equal to the form of `*x`. `_` can be used for anonymous `''ident` type variables. In this document, such restriction does not apply if the type variable is not prefixed with any quote character.

`num` is for floating point numbers. `bool` is for boolean values. `keyword` is a pair of a small integer and a string identifier. All other primitive types are defined by the user using the foreign function interface.

For built-in data colletion types, we have a total of 7 of them.

syntax	specification
`t1,t2,..,tn`	tuples; for empty tuples `()` is used
`(|field:t1; field2:t2 | rowtype|)`	anonymous records with structural typing, and it can be seen as a named version of tuple
`[typexpr]`	immutable lists
`[|typexpr|]`	immutable and persistent arrays
`?typexpr`	optionals
`t1\t2\..\tn`	anonymous tagged unions
algebraic data types	supported with the difference to OCaml that each data constructor must takes exactly one argument.

`*typexpr` is a stream, in which each element is of `typexpr`. `(n*)typexpr` is equivalent to `**..**typexpr`, where the number of leading stars is `n`.

`t1->t2` is a function with an argument `t1`. Multi-argument functions are curried, as with other Hindley-Milner type systems.

2.1 Zero value

For certain type `t`, a zero value `zero t` could be created.

`t`	`zero t`
`num`	`0.0`
`bool`	`false`
`t1,t2,..,tn`	`(zero t1),(zero t2),..,(zero tn)`
`(|f1:t1;f2:t2,..|)`	`(|f1=zero t1;f2=zero t2;..|)`
`[t]`	`[]`
`[|t|]`	`[||]`
`?t`	`??`
`t1\t2\..\tn`	`(zero t1)\\..\`
`t1->t2`	`fun _ -> zero t2`
`*t`	`*(zero t)`
some of external types	return value of C function `makezero()`

2.2 Stream generalization

Sometimes we hope a function `f : a -> b` can also work on any nested levels of streams, i.e. `*a->*b`, `**a->**b`, etc. We could use stream generalization to make a function so.

Only values of certain types can be stream-generalized. A value `f` of type `(n1*)'a -> (n2*)'b -> (n3*)'c -> d` can be stream-generalized with respect to arity 3, if `n1`, `n2` and `n3` are all constants. Then the stream-generalized value `fgen` have the type `(n*)(n1*)'a -> (n*)(n2*)'b -> (n*)(n3*)'c -> d`, and its definition depend on the type parameter `n`. In the code below, the number of nested `dew ->` is equal to `n`.

```
let fgen a b c = dew -> let a1=@a and b1=@b and c1=@c in
                   dew -> let a2=@a1 and b2=@b1 and c2=@c1 in
                     ...
                       dew -> let an=@an_1 and bn=@bn_1 and cn=@cn_1 in
                         f an bn cn
```

Builtin constructs are as stream-generalized as possible, with the details as follows:

builtin constructs	stream generalized
constants, e.g. `1.23`	yes
data cosntructors, e.g. `?x`	the constructed data is generalized with arity 0
record and array update	the constructed data is generalized with arity 0
anonymous functions	no
arithmatic and logic operators	yes
relational operators	yes except `===` and `!==`
`<:`, `apop`, `alen`	only `alen`
selectors	no
stream operators	no

3 Expressions

```
expr ::= ident
       | dynamic-ident
       | constant
       | expr {, expr}+
       | (| recfield {; recfield} [;] |)
       | (| expr with recfield {; recfield} [;] |)
       | expr :: expr
       | [ expr {; expr} [;] ]
       | [| expr {; expr} [;] |]
       | [| expr with ( expr ) = expr {; ( expr ) = expr} [;] |]
       | ? expr
       | {\}+ expr
       | expr {\}+
       | {\}+ expr {\}+
       | zero
       | unop expr
       | expr binop expr
       | expr {relop expr}+
       | expr selector
       | alen expr
       | apop expr
       | expr {expr}+
       | let let-binding { and let-binding } in expr
       | if expr then expr [else expr]
       | match expr with pattern-matching
       | while expr do expr done
       | for pattern = expr (to | downto) expr [step expr] do expr done
       | for pattern in expr do expr done
       | fun {pattern}+ [: typexpr] -> expr
       | assert expr
       | ident {selector} = expr
       | ident {selector} assign-op expr
       | expr ; expr
       | ( expr )
       | ( expr : typexpr )
       | begin expr end
       | pre expr expr
       | * expr
       | @ expr
       | dew -> expr
       | dew' -> expr
       | switch expr
       | keepalive expr expr
       | async external-func

recfield ::= ident [: typexpr] [= expr]
recfield-pattern ::= ident [: typexpr] [= pattern]

let-binding ::= pattern = expr
              | (ident | dynamic-ident) {pattern}+ [: typexpr] = expr

pattern-matching ::= [|] pattern [when expr] -> expr
                     {| pattern [when expr] -> expr}

pattern ::= ident
          | dynamic-ident
          | _
          | constant
          | pattern as (ident | dynamic-ident)
          | pattern | pattern
          | pattern { , pattern }+
          | (| recfield-pattern {; recfield-pattern} [;] |)
          | pattern :: pattern
          | [ pattern { ; pattern } [;] ]
          | [| pattern { ; pattern } [;] |]
          | ? pattern
          | {\}+ pattern
          | pattern {\}+
          | {\}+ pattern {\}+
          | constr pattern
          | ( pattern )
          | ( pattern : typexpr )
          | * pattern

selector ::= . ident
           | .( expr )
```

```
constant ::= number
           | true
           | false
           | ()
           | (||)
           | []
           | [||]
           | ??
           | keyword-ident

unop ::= -
       | !

binop ::= + | - | * | / | // | % | **
        | && | ||
        | ::
        | <:
        | $

relop ::= < | <= | > | >= | == | != | === | !==

dynamic-ident ::= % ident
keyword-ident ::= $+ ident
tuple-ident ::= _ integer
```

3.1 Evaluation semantics

Each stream is evaluated exactly once per frame and maintains a result cache during its lifetime. A stream's lifetime can end at either the end of a frame or the beginning of a frame. In both cases, the stream is dropped at the end of that frame; while the stream's current value is only evaluated in that frame for the former case.

The lifetime of a stream is largely determined by its garbage collection reachability from any garbage collection root. In Dew, garbage collection reachability is defined as the literal program. For example, the stream `dew -> let _ = a in ()` always holds the reference of `a`, even it's not used in the stream.

The detailed rules of a stream's lifetime are as follows:

1. If a stream is reachable from a garbage collection root at the end of a frame, its lifetime is extended to the beginning of next frame.
2. If a stream's current value is needed in a frame, its lifetime is extended to the end of the frame.
3. If an early stream is alive at the beginning of a frame, its lifetime is extended to the end of the frame.

The evaluation order is undefined in many cases, but otherwise takes the usual order of strict evaluation. Cases of undefined evaluation order are:

1. The evaluation order of different streams is undefined;
2. The evaluation order of a stream and the context from which the stream is created;
3. The evaluation order of multiple parallel definition expressions in local `let` expressions.

Undefined evalution order presents great opportunities for parallel computation, although it is undefined whether the compiler actually takes the opportunity or not. Dew programs is executed within one system thread with multiple logical threads, and asynchronous function calls are dispatched to an external job queue in core engine.

It is not guaranteed that no dependency cycles of streams exist when the program compiles. When defining a stream-typed variable, the definition of the variable is allowed to depend on the variable itself, either directly or indirectly. As a specific example, `let x = f x` is valid. To prevent dependency cycles, the implementation of a function may need to cooperate with the function's uses at client sites. Dependency cycles encountered at run-time will be detected and reported, rather than resulting in infinite loops.

3.2 Non-stream-specific expressions

It's mostly the same with OCaml, with some modifications.

1. In contrary to OCaml, values are never mutable, but local variable bindings are always mutable. Mutable bindings can be later re-assigned with `ident = expr`. When a mutable binding is used in a function or stream closure, the value at the moment that the closure is created is used. For many operators, `ident = ident op expr` can be written as `ident op= expr`.

   You may not be able to do assignments on the right hand side of `let` expressions in certain cases, because expressions on right hand side of `let` may have to be turned into thunks for proper evaluation.

2. The `let ... in ...` expression is recursive by default, unlike OCaml.

3. The usual generalising behaviour of `let ... in ...` in Hindley- Minler systems is dropped. It adds complexity to type inference, while does not provide much benefit. Module-level values not defined with a function `let` binding are also not generalized.

4. Dynamically scoped variables are introduced, and written as `%ident`. They are used to implicitly pass context information down for a whole block of dynamical code scope. A dynamically scoped variable must be initialized with a module-level `let` before it can be used. When a dynamically scoped variable is used in a `let` binding, the `let` binding would be made non-recursive.

5. Keywords written like `$$ident` are introduced. They used to efficiently represent string identifiers or symbols, and have the type `keyword`. Besides the string identifier part, keywords are also associated with a level number, which is the number of leading `$` characters subtracted by one. Possible level number is in the range `0..7` inclusive.

6. Special syntax for optional values are introduced. `?a` stands for `Some a`, and `??` stands for `None`.

7. Consecutive comparison like `a <= b <= c` is added. Also, OCaml equality test operators `=` and `<>` are changed to `==` and `!=`; a boolean not operator `!` is also added.

8. The `for ... in ...` expression has been added to iterate through all elements from a list.

9. A `zero` primitive has been added to create a zero value for the type. `zero` is generative, and each use of `zero` creates a new value.

10. Primitive operators are stream-generalized, e.g. `+` has the type `(*n)num -> (*n)num -> (*n) num`. This does not include data constructors like `::`, because data constructors doesn't have a constant nesting level for their arguments.

11. External function can be called asynchronously via `async fn`. Asynchronous external function calls are scheduled in parallel in multiple OS threads. The external function `fn` can also be a stream-generalized, in this case each of the external calls is async.

12. Arrays are immutable unlike OCaml's. `num`-typed indexes are rounded to the nearest integer before looking into the array, and out-of-bounds access are reported.

13. We introduce the notion of selectors, which could be used to get or set data within deeply nested structures.

14. Data constructors for ADTs are used as normal functions in an expression, instead of requiring a special syntax.

15. Instead of `@@` for function application operator, we use `$` from Haskell.

Lexical syntax also have some changes from OCaml:

1. Whitespace is significant. If a operator is separated from its operands by whitespace, it's a loose operator; or if it's immediately close to its operands, it's a tight operator. Tight operators bind more tightly than function application, which in turn binds more tightly than loose operators. This helps to remove parentheses around negative numbers, make unary operators have correct precedence than binary operators, and other benefits.

2. Comment is started by character `#`, and continues to the end of line. Single-character comment is decided to be important to this language used for scripting.

3.3 Stream-specific expressions

`pre`, `*`, `@` are three primitive operators to build static networks of streams. `dew ->` is the primitive to build dynamic stream networks. `switch` and `keepalive` are primitives to control a stream's lifetime.

`pre e1 e2` creates a stream at the time it is evaluated, which first value is `e1`, and each subsequent values is the previous one from `e2`. `*e1` creates a stream, which repeats the value `e1` for ever. `@e1` takes the current value of stream `e1`.

`dew ->` is used to create dynamic stream networks. `dew -> e1` evaluates `e1` at each time, and collecting all results into a stream of type `*typeof(e1)`.

`dew' -> e1` is similar to `dew -> e1`, and also produces a value of type `*typeof(e1)`, except that the spawned stream is marked as an early stream. An early stream that is spawned before the current frame is guaranteed to be scheduled before a non-early streams, if the early stream's evaluation doesn't depend on it. This is intended to get initial engine state of a frame, which engine state may be modified if queried later.

`switch` has the type signature of `*(a\*a)->*a`. `switch s` makes the output of stream `s` dynamically switchable to a new stream. When the current value of `s` turns to `\s'`, the reference to `s` will dropped at the end of frame; and from the next frame on, the stream `switch s` would switch to return the value from the stream `s'` instead. Two adjacent `switch` are guaranteed to be collapsed, and thus keeping the memory and time cost bounded over time when doing consecutive `switch`s.

`keepalive` has the type signature of `*bool->a->()`. `keepalive b s` creates a garbage collection root pointing to `s`. The lifetime of the garbage collection root is controlled by `b`, once `b` turns `false`, the garbage collection root is dropped, and the effect is visible at the end of frame.

Pattern matching is extended for streams with a special semantics. `*pat` matches a value `s` of type `*t`, if `pat` matches values of type `t`; and for any value of `*t`, it's always accepted for pattern `*pat`. The actual pattern matching of `pat` against values from the stream `s` occurs in a constructed stream, and potentially last forever. If `x` is a bound variable in `pat` with type `t`, `x` would also be a bound variable for `*pat`, with type of `*t` and value of

```
dew -> match @s with pat -> x
```

4 Module-level definitions

```
definition ::=
   | import relative-path [as ident]
   | extern type ident = exttype-storage { exttype-attr }
   | extern (func | func') ident : c-funtype = c-ident { extfunc-attr }
   | type-definition
   | (let | let') let-binding { and let-binding }

c-funtype ::= [instdata ->] {c-type ->}+ c-type

exttype-storage ::= byref c-ident c-ident | byvalue integer

exttype-attr ::= [ zero c-ident ]
               | [ serialize c-ident c-ident ]
               | [ equal c-ident ]
               | [ compare c-ident ]
```

```
extfunc-attr ::= [ pure ]
               | [ readonly ]
```

For clarity, `relative-path` and `c-ident` are wrapped with double quotation marks (`""`).

The verison of `let'` and `extern func'` has the difference to the regular version that the defined function is stream-generalized.

An expression can be placed among module-level defintions to be evaluated for side effects. The type of such expressions must be `(n*)()`.

4.1 Module system

A single file is a module. `import "path/to/module"` imports module-level identifiers from `"path/to/module"` into current module, except for those starting with an underscore (_) character, which indicate the identifier is private to that module. Field names and dynamically-scoped variables that starts with _ also won't be imported. `import "path/to/module" as mod` imports the bound identifier `ident` as `mod_ident`. It is always reported if a name collision happened during the importing.

Cyclic module dependencies are not allowed. For each compilation unit, a single root module is specified by the user, and the modules it depends on are also compiled in. Module-level definitions and expressions serve as entrypoints, and would be evaluated when the script is started.

Dew program filenames are suffixed with ".dew", and the suffix can be omitted for references to the module.

4.2 Foreign function interface

`extern` introduces foreign function interface functionalities, which is crucial to communicating with the core engine. For both external types and external functions, some attributes can be specified.

The storage of an external type must be specified to be either `byref incref decref` or `byvalue size`. `byref incref decref` types are represented by a non-null C pointer, and memory-managed by reference counting. An extra reference count must be hold for passing between Dew program and core engine. Functions `incref` and `decref` are reference count incrementer and decrementer of the type, and have the prototypes of:

```
void incref(void* ptr);
void decref(void* ptr);
```

byref external types are allowed to have interior mutability to work with the game engine.

byvalue size external types are represented by a binary blob, which interpretation is opaque to Dew. size is the size (in bytes) of the binary blob.

4.2.1 Data conversion between C and Dew

c-types specifies how the value is converted between Dew program and the core engine. If there are multiple return values, the ones except the last is passed via a pointer argument. instdata can only appear at the beginning of cfuntype; and is Dew instance specific data, which is given when starting the dew instance. keyword is represented by a uint64_t in C, in which the 3 least-significant bits represent the keyword level, and the rest bits represent the keyword string hash. Lists and are represented as a C array when transfered across the Dew program border.

c-type or instdata	Dew type	C type
real	num	double or float
int32	num	int32_t
int64	num	int64_t
keyword	keyword	uint64_t
bool	bool	bool
byref external	external-type	non-null void*
?byref external	?external-type	nullable void*
byvalue(n) external	external-type	n-sized struct
[c-type]	[Dew-type]	int32_t, void*
()	()	(nothing)
t1,t2,..,tn	t1,t2,..,tn	t1,t2,..,tn
instdata	(nothing)	void*

External reference-counted types follow the general convention: the callee borrows references from the caller as arguments, but returns owned references. The C array for lists are memory-managed in a similar way: the callee gets borrowed C arrays from the caller, but returns owned C arrays.

4.2.2 Attributes for external datatypes and functions

An external type attribute can be of the following:

1. `[zero makezero]`. This external type has a zero value associated, which could be obtained by calling `makezero`.

   ```
   void* makezero();
   ```

2. `[serialize toblob fromblob]`. This type can be serialized into a binary blob.

   ```
   char* toblob(void* ptr);
   void* fromblob(char* blob);
   ```

3. `[equal f]`. This type can be compared for equality, and can be compared with `==` and `!=` in Dew programs.

   ```
   bool equal(void* ptr1, void* ptr2);
   ```

4. `[compare f]`. This type is totally ordered, and can be compared with any of the relational operators in Dew programs.

   ```
   int compare(void* ptr1, void* ptr2);
   ```

An external function attribute can be of the following:

1. `[pure]`. This external function always returns the same result for the same input. The inputs and results are same in the sense that they are equivalent from the view of application.

2. `[readonly]`. This external function only read engine state, and does not modify engine state.

5 Syntax sugars

Syntax sugars work like macros, and specific to my work-in-progress game. This approach is used, because: (1) I can't afford to add macro facilities to the language, which would be costly; (2) some syntax forms are difficult to achieve with usual macro features.

5.1 Geometry type and operators

```
geom-typexpr ::= vec

geom-expr ::= %- vec-expr
            | vec-expr %+ vec-expr
            | vec-expr %- vec-expr
```

```
          | vec-expr %* num-expr
          | vec-expr %/ num-expr
          | vec-expr %. vec-expr
          | vec-expr %x vec-expr
```

The type `vec` represents a 2D point or vector, and is equivalent to a tuple of two numbers `(num, num)`.

Operators `%.` and `%x` are dot product and cross product respectively.

5.2 Curve specifier

```
curve-expr ::= element {connector element}+
element ::= expr
          | [ expr ]
          | cycle
connector ::= ..
            | --
```

The syntax is intended to be similar to MetaPost. How many curve control parameters we support, and what the full syntax is like are not decided yet.

6 Standard library

Basic operators `pre`, `*` and `@` can be seen to operate on the outmost level of streams. We can also define a set of functions to operate on inner levels of the streams, and then stream-generalize that function.

These set of stream functions are parameterized with a stream level `n` to operate on. The funtions have the following types before stream-generalize.

n	pren	repeatn	currn
1	`'a -> *'a -> *'a`	`'a -> *'a`	`*'a -> 'a`
2	`*'a -> **'a -> **'a`	`*'a -> **'a`	`**'a -> *'a`
3	`**'a -> ***'a -> ***'a`	`**'a -> ***'a`	`***'a -> **'a`
…	…	…	…

We also define a set of functions for application of nested streams of functions.

```
app1 f x = dew -> let f = @f and x = @x in f x
app2 f x = dew -> let f = @f and x = @x in app1 f x
```

```
app3 f x = dew -> let f = @f and x = @x in app2 f x
...
```

7 Compiler usage and C API

```
dewc <root-module>
     -n <compilation-unit-name>
     -o <llvm-bytecode>
     [-numtype <float or double>]
     [-g]
```

Each invocation of `dewc` compiles a single compilation unit, which would output LLVM bytecode. The LLVM bytecode could then be compiled into `.so` files, and loaded dynamically into the running game.

```
typedef struct {
   void* (*malloc)(size_t size);
   void* (*realloc)(void* ptr, size_t size);
   void (*free)(void* ptr);

   void (*job_spawn)(void (*func)(void*), void* data);

   size_t size_minorheap;
   size_t size_errormsg;
} dew_config_t;

typedef struct {
   int minorheap_max;
} dew_stats_t;

typedef struct dew_def dew_def;
typedef struct dew_unit dew_unit;

void dew_setup(const dew_config_t* config);
dew_unit* dew_start(const dew_def* dewdef, void* instdata);
bool dew_step(dew_unit* dew);
void dew_stop(dew_unit* dew);
void dew_error(const char* fmt, ...);
const char* dew_geterror(dew_unit* dew);
const dew_stats_t* dew_getstats(dew_unit* dew);
void dew_debug_register(const dew_def* dewdef);
void dew_debug_unregister(const dew_def* dewdef);
const char* dew_debug_kwstr(uint64_t kw);

extern dew_def dew_def_CUN;
```

`CUN` stands for compilation unit name, and it would be replaced with real names for each Dew compilation unit. Each compilation unit would export a single symbol `dew_def`; all other functions are exported from the Dew runtime library.

`dew_setup` must be called to initialize the entire Dew subsystem, before any other function can be called. The function `dew_start` only starts the dew unit in a suspended state, and does not evaluate it for the first frame. Each `dew_unit` is independent from other `dew_unit`s, and different `dew_unit`s can be safely run on different system threads in parallel. However, nested `dew_unit` evaluation is not supported, i.e. `dew_step` is not reentrant.

`dew_error` can be used to report an error in either synchronous or asynchronous external function calls from Dew. However, care must be taken not to leave the core engine in an inconsistent state, if you want the game continue running.

Either `dew_error` or errors within the Dew script would make the Dew instance in an error state. A Dew instance in error state can be destroyed with `dew_stop`; and make sure `dew_stop` is called from the same system thread as the first failed call of `dew_step`.

`dew_debug_register` and `dew_debug_unregister` can be called to make debug information available. Debug information include stacktraces and keyword strings, which are fetched via `dew_debug_kwstr`.