Build game scripts DSL's with the Metacasanova metacompiler

Francesco Di Giacomo

Università Ca' Foscari di Venezia

Reasons behind scripting DSL's

Games contain very complex behaviours:

- Wait to be close enough to interact with an item.
- Perform an action only if the key was pressed and then released.
- Execute two tasks in parallel and take the result of the one that terminates first.
- Prioritized behaviours.

Reasons behind scripting DSL's
Examples

Games contain very complex behaviours:

- Interacting with a door only if you are close to it.
- Shoot with a handgun
- A special moves in a fighting game: we want that pressing a key in the combination is done within a given time.
- A guard AI that patrols with lowest priority, shoot the enemy with medium priority, and take cover with highest priority.

Reasons behind scripting DSL's
Advantages of a DSL

- These behaviours are hard to express in GPL's because they require constructs not provided within the language.
- Wait for a certain amount of time, wait for a condition, concurrency operators, priority operators, ...
- We would really like to be able to write `wait 5.0f` in our code.

Implementing DSL's for games

- Possible hard-coded solutions: strategy pattern, monadic coroutines, generators (virtuality involved).
- State machines, compiler for a custom scripting DSL's (better performance).
- It is possible to create a multi-threaded game engine, but impossible to create a thread to handle each one of the situations above.

Compilers are a very popular choice

- Warcraft III: Just Another Script Syntax (JASS)
- Starcraft II: Galaxy Script
- ArmA series: Status Quo Scripts (SQF).
- Neverwinter Nights: NWN Script.
- Unreal Engine: UnrealScript.

```
"colorCorrections" ppEffectAdjust [1, pi, 0, [0.0, 0.0, 0.0, 0.0], [0.05, 0.18, 0.45, 0.5], [0.5, 0.5, 0.5, 0.0]];
"colorCorrections" ppEffectCommit 0;
"colorCorrections" ppEffectEnable true;

thanatos switchMove "AmovPpneMstpSrasWrflDnon";
[[],(position tower) nearestObject 6540,[["USMC_Soldier",west]],4,true,[]] execVM "patrolBuilding.sqf";
playMusic "Intro";

titleCut ["", "BLACK FADED", 999];
[] Spawn
{
  waitUntil{!(isNil "BIS_fnc_init")};
  [
    localize "STR_TITLE_LOCATION" ,
    localize "STR_TITLE_PERSON",
    str(date select 1) + "." + str(date select 2) + "." + str(date select 0)
  ] spawn BIS_fnc_infoText;
  sleep 3;
  "dynamicBlur" ppEffectEnable true;
  "dynamicBlur" ppEffectAdjust [6];
  "dynamicBlur" ppEffectCommit 0;
  "dynamicBlur" ppEffectAdjust [0.0];
  "dynamicBlur" ppEffectCommit 7;
  titleCut ["", "BLACK IN", 5];
};

sleep 5;
```

Figure: Code snippet from SQF script

Compilers are complex

- Compilers are very complex pieces of software.
- Parser, Type checker, intermediate code generation, interpreter or assembler.
- Expensive, hard to add extra features to the language.

Compilers are repetitive

1. Write a formalism for the grammar of the language.
2. Build a parser for the syntax analysis based on the grammar.
3. Write a formalism for the type system.
4. Build a type checker for the type analysis based on the type system.
5. Write the semantics of the language
6. Generate code according to the semantics.

Compilers are repetitive

- The only "creative" part of the process is step 1,3,5 (on the paper)
- The rest is just translating those points in the chosen language.
- What if it was possible to have a compiler that accepts as input a language definition, a program written in that language, and outputs executable code?

Idea behind metacasanova

- Write the language definition (type system and semantics) as a real program.
- Generate code according to this definition.
- Advantage: no need to encode the definition in a programming language (no hard-coded compiler).

Overview of Metacasanova

- **Data declaration**: used to represent the syntactical constructs of the language (meta-data).
- **Function declaration**: They define the meta-types (meta-types transformations).
- **Rules**: They define how the semantics of the syntactical constructs (language semantics)
- **Subtyping**: They define equivalence among meta-types

Examples

Data declarations:

```
Data Expr -> "+" -> Expr : Expr  Priority 500
Data "$f" -> <<float>> : Value Priority 10000
```

Function declarations:

```
Func "eval" -> Expr : Evaluator => Value
```

Rules:

```
eval a => $f c
eval b => $f d
<<c + d>> => res
-----------------------------
eval (a + b) => $f res

------------------------
eval ($f f) => $f f
```

Subtyping:

```
Value is Expr
```

- **Entities:** They contain both the data and the behaviour of the objects in the game
- **Rules:** They define the behaviour of the entity. Once a rule ends its execution it is restarted at the next frame.
- **Domain:** A set of entity attributes the rule is allowed to change. A rule can always read all fields but can modify only those in the domain through a `yield` statement.

Example of program in Casanova

```
entity Ball = {
Position   : Vector2
Velocity   : Vector2

rule Position = Position + Velocity * dt

rule Velocity =
        wait Position.X >= 300f || Position.X <= 0f
        yield new Vector2(-Velocity.X, 0f)
}
```

Rule pausing

Rule execution can be paused with built-in statements:

- `wait` takes either a floating point value or a predicate. In the first version the rule is paused by the given amount of seconds, in the second it is paused until the condition is met.
- `yield` updates the fields in the domain with the given values. The rule execution is paused by one frame in order to be able to see the changes at the next game update.

Semantics of wait

$$\frac{\langle t - dt > 0 \rangle \Rightarrow \texttt{true}}{\langle \texttt{wait } t; k\ dt \rangle \Rightarrow \langle \texttt{wait } t - dt; k\ dt \rangle} \qquad \frac{\langle t - dt > 0 \rangle \Rightarrow \texttt{false}}{\langle \texttt{wait } t; k\ dt \rangle \Rightarrow \langle k\ dt \rangle}$$

$$\frac{\langle c \rangle \Rightarrow \texttt{true}}{\langle \texttt{wait } c; k\ dt \rangle \Rightarrow \langle k\ dt \rangle} \qquad \frac{\langle c \rangle \Rightarrow \texttt{false}}{\langle \texttt{wait } c; k\ dt \rangle \Rightarrow \langle \texttt{wait } c; k\ dt \rangle}$$

Implementation of wait in Metacasanova

In the following waiting on a condition is called `when` because Metacasanova does not currently support operation overloading

```
eval expr ctxt => ($f t)
<<t <= dt>> == false
-----------------------------------
eval_s (wait expr) k ctxt dt => Suspend (wait $f (<<t - dt>>));k ctxt

eval expr ctxt => ($f t)
<<t <= dt>> == true
-----------------------------------
eval_s (wait expr) k ctxt dt => Resume k ctxt


eval expr ctxt => ($b true)
-------------------------------------------
eval_s (when expr) k ctxt dt => Atomic k ctxt

eval expr ctxt => ($b false)
----------------------------------------
eval_s (when expr) k ctxt dt => Suspend (when expr);k ctxt
```