Build game scripts DSL's with the Metacasanova metacompiler

Francesco Di Giacomo [1]
Mohamed Abbadi [1]
Agostino Cortesi [1]
Pieter Spronck [2]
Giuseppe Maggiore [3]

Università Ca' Foscari di Venezia

Tilburg University

Hogeschool Rotterdam

Games contain very complex behaviours:

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

Games contain very complex behaviours:

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

- 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.

- 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 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.

- The only "creative" part of the process is step 1,3,5 (on the paper)
- The rest is just about implementing 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?

- 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) $\Rightarrow$ more scalability (just copy/paste the formal definition).

- **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.

```
entity Guard = {
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 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

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

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

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

- Script making a guard patrol two checkpoints
- Tests run with the script written in Casanova 2.5 and Python.
- Implementation of Casanova 2.5 compared with the code length of the hard-coded compiler modules for Casanova 2.0.
- Python is used as a scripting language in several games (e.g. World in Conflict, Civilization IV).
- We neglected C# because we already have benchmarks for that in Casanova 2.0 and both languages are faster of several orders of magnitude.

Results: patrol script

Casanova 2.5		
Entity #	Average update time (ms)	Frame rate
100	0.00349	286.53
250	0.00911	109.77
500	0.01716	58.275
750	0.02597	38.506
1000	0.03527	28.353
Python		
Entity #	Average update time (ms)	Frame rate
100	0.00132	756.37
250	0.00342	292.05
500	0.00678	147.54
750	0.01087	91.988
1000	0.01408	71.002

Casanova 2.5 with Metacasanova	
Module	Code lines
Data structures and function definitions	40
Query Evaluation	16
While loop	4
For loop	5
If-then-else	4
When	4
Wait	6
Yield	10
Additional rules for Casanova program evaluation	40
Additional rules for basic expression evaluation	201
Total: 300	
Casanova 2.0 compiler	
Module	Code lines
While loop	10
For-loop and query evaluation	44
If-Then-Else	15
When	11
Wait	24
Yield	29
Additional structures for rule evaluation	63
Structures for state machine generations	754
Code generation	530
Total: 1480	

Table: Patrol sample evaluation

Conclusion

Advantages:

- The code length to define the semantics of the language is almost 5 times shorter.
- Using Metacasanova greatly reduces the effort to build a compiler.
- The definition of the language in Metacasanova is almost identical to the formal definition.
- Extending the language becomes easier.

Disadvantages:

- The generated code is slower than Python but of the same order of magnitude.
- Code performance affected by maps used to model the memory.
- The code generation requires further improvement.
- Using data structures and methods from external libraries requires to build a wrapper with meta-types and meta-functions.

Work in progress/Future works

- Implement type functions to generate modules at compile time
- Inline the result of the evaluations of type functions directly in the emitted code.
- It should solve both the problem related to the memory model and the need for a wrapper for external data structures.
- Extend Casanova 2.5 with networking primitives.

Tank you!

Questions?