Lazy waiting: an optimized data-flow compiler for Casanova

1 Introduction

Computer and video games have a large market which has grown to the point that the sales are higher than those of the music and film business [**?**]. Sales of mobile games alone, have been predicted to exceed 100 billions of dollars in 2017 [**?**]. The adoption of games has changed to the point that they are not used only in the entertainment business, but they have also found applications in education, training, research, social interaction, and even raising awareness. These so-called *serious games* are used in schools, hospitals, industries, and by the military and the government. Researchers use games to simulate and evaluate their results. Examples of successful serious games are: Microsoft Flight Simulator, VBS1, Second Life, and Phylo [**?**]. These games do not enjoy the same rich market of entertainment games, but their social impact is nevertheless high.

The core of a computer game is composed by the *state*, which is a data structure storing all the information about the simulation, and the *game loop*, that is a function which loops indefinitely and calls two procedures: *(i)* the update function which updates the game state according to the *logic* of the game, and *(ii)* the draw function which draws the scene. Each iteration of the game loop is called *tick* or *frame*.

The update function usually updates the state in two different ways: *(i)* continuous dynamics, describing the physics behaviour of an entity, and *(ii)* discrete dynamics, when the update is not made at any instant but just when some conditions are met, often describing interactions between characters (i.e. their behaviour), the *Artificial Intelligence*, events happening in the game world.

Discrete dynamics requires complex concurrent, nested, parallel, interruptible programs. These programs are typically hand-crafted, with little to none language support. This is a costly (and error-prone) activity. Thus, we argue that there is a need for language constructs that are capable of expressing the discrete dynamics of game logic while , at the same time, maintaining fast runtime. The aim of our work is the definition of such constructs and their optimization, taking advantage of the "suspensive" nature of discrete dynamics.

Problem statement In this paper we will focus on two main tasks centred around discrete dynamics.

1. Define a series of concurrent operators to describe discrete dynamics in games.

2. Define an optimized concrete semantics for these operators to ensure their runtime costs are acceptable for *Real Time Games*.

We will show that the use of these operators reduces the effort of building, maintaining, and optimizing discrete dynamics code employed in computer games. Our work focuses on tools and techniques to reduce complexity and improve performance. Developers are only asked to focus on the definition of the game with domain-appropriate operators, making it readable and leaving optimizations to the compiler.

1.1 Structure of the paper

We start with a discussion about games and their complexity, Through the use of a case study (Section 2), we identify the issues of discrete dynamics implementation in traditional languages. We show the operators and the main concept behind Casanova 2 (Section 3), a declarative-imperative programming language oriented to video game development. We introduce the idea of *lazy waiting*, a hybrid static-dynamic analysis technique which allows us to optimize our operators runtime costs. (Section 4). We explain in details the syntax and semantics of our operators and provide a detail explanation of the implementation of lazy waiting (Section 5). We show the effectiveness of our approach with the evaluation, by giving an implementation of our case study using our optimized operators in Casanova 2, and comparing it with alternative implementations (Section 6).

2 Case study

We now show an example of discrete dynamics, used as a case study throughout this paper. Let us consider a game where a guard patrols a room. A set of enemies will periodically enter the room and try to walk through the guard post. The guard is given a set of waypoints which will form the patrol route. A healer will periodically visit the guard post to offer healing services.

The behaviour of the guard can be re-defined as a set of prioritized tasks, listed by descending priority, as follows:

1. Health $< 0 \Rightarrow$ The guard is dead.
2. Low health $\Rightarrow$
   (a) Enemy near $\Rightarrow$ retreat
   (b) Healer available $\Rightarrow$ Go to the healer and recover.
   (c) Otherwise $\Rightarrow$ Do nothing.
3. An enemy is in range $\Rightarrow$ Attack the enemy.
4. Otherwise $\Rightarrow$ cycle through waypoints and slowly recover health.

These tasks can be modelled by the *Behaviour Tree* shown in Figure 1. In this representation we used the notation $\emptyset$ for the root node of the tree, and ? for *fallback nodes*. A fallback node will return immediately when the first of its children succeeds, ticking them from left to right.

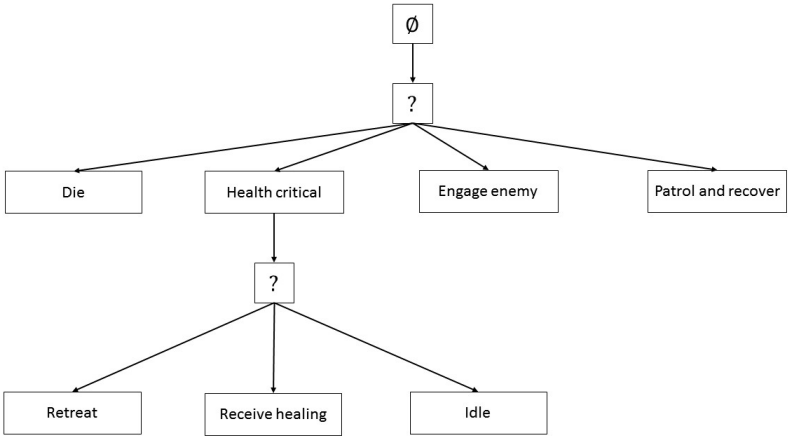

Figure 1: Guard behaviour tree

Note that, according to the priority rule defined above, if there is an enemy near the guard but his health is critical (i.e. below a specified threshold), he will run a retreat/heal subtask.

This example illustrates how even a simple behaviour, such as that of the guard, requires very complex control structures which must be implemented manually by the programmer. From the case study the following behavioural patterns emerge:

- *Waiting*: used during the self-healing of the guard while patrolling, since the health must increase only after an interval of time has elapsed.
- *Concurrency*: used for choosing which task the guard must accomplish.
- *Parallelism*: used during the guard patrol, when he moves to the next waypoint and at the same time he recovers health periodically.
- *Pre-emption*: used to abort a task when another task with higher priority is ready to be executed.
- *Nesting*: used when building sub-tasks for the guard.

If these patterns are not well supported by a programming language, they must be rebuilt every time. This requires to manually manage the coordination between states, and increases the chance of errors in the code. Although derived from this case study, we argue that this operators are general enough to cover the most common discrete dynamics used in games, by combining them appropriately.

In the following section we will give a brief introduction to the Casanova 2 language, a declarative-imperative programming language oriented to the development of video games. In Section 4, as anticipated above, we will show the

definition of a set of discrete dynamics operators (based on the patterns described in the case study) and then introduce an optimization technique called *lazy waiting* to reduce the overhead due to the waiting behaviour of these operators.

3 Casanova overview

Casanova 2 is a declarative-imperative programming language oriented to video game development. A Casanova program is a set of `entity` definitions organized in a tree structure. The root is a unique entity called `worldEntity`. Each entity is made of: (*i*) a set of *fields*, *(ii)* a set of *rules*, and (*iii*) a *constructor*.

A field can have a primitive type (supported types are the most common in other programming languages, such as `int`, `float`, `string`, *tuples*, *lists*) or another entity type. A branch in the tree structure occurs when one field is declared having an entity type. The following example shows a field declaration

```
entity E = {
   A : E1
   B : float
   C : int
   ...
}
```

Rules are defined on one or more fields with the keyword `rule`. A rule can modify the value of fields by using the statement `yield` on the fields it take as arguments. Besides rules are also passed as implicit arguments the following three values: a reference to the `worldEntity`, a reference to the current entity (`this`), and a floating point value (`dt`) which contains the time difference between the current frame and the previous one. Each rule executes its body continuously, i.e. once the execution has reached the end of the rule body it is resumed from the beginning. The following example shows a rule definition:

```
rule P,V =
   let new_v = V + g * dt
   let new_p = P + V * dt
   yield new_p,new_v
```

A constructor is declared using the keyword `Create` and may take some parameters as arguments. It returns the initialization of the entity fields enclosed by curly brackets[1]. The following example shows a constructor definition:

```
Create(x : E1) =
   {
      A = x
      B = 1.0
      C = 0
      ...
   }
```

Casanova 2 supports the most common control structures such as `if-then-else`, `while-do`, `for`, and also SQL-like queries such as (`for x in list - select - where`). Local variables can be declared using the `let` statement.

[1]This because Casanova 2 syntax is based on F# syntax and entities are treated as F# records, thus a constructor returns a record initialization.

4 Concurrent operators

The case study presented in Section 2 would require to use very complex synchronization conditions and discrete dynamics to be implemented. To simplify the treatment of discrete dynamics, we propose to describe them through the following operators:

- **Waiting** (`wait`) takes as input either a floating-point value or a boolean expression. If a floating-point value is provided, then the operator stops the execution of the function for the specified amount of time. If a boolean condition is provided then the execution is stopped until the condition is met.
- **Parallelism** (`.&`) takes as input multiple code blocks and keeps executing them indefinitely.
- **Concurrency** (`.|`) takes as input multiple conditions, each associated to a code block. Whenever one of these conditions is met, it executes the corresponding code block associated to it, ignoring all the others.
- **Pre-emption** (`!|`) takes as input multiple conditions, each associated to a code block. The conditions are prioritized, i.e. if multiple conditions are met, the code associated to the topmost one is executed. Furthermore, the code blocks are interruptible, which means that, if a condition with a higher priority becomes true, the execution of the current block is stopped and it will resume at the beginning of the block associated with the higher priority.

Our case study can be implemented using these operators. In the following example we assume that `Guard` has two fields `NearThreats` and `TargetsInRange` that contain respectively a list of close enemies to the guard when he is critically wounded, and a list of targets which can be attacked by the guard.

```
entity Guard = {
  //field declarations
  //...

  rule Target,Target.Health,this.Health =
    !| this.Health <= 0 =>
        //play dead animation
    !| this.Health <= this.HealthThreshold =>
        !| this.NearThreats.Length > 0 =>
            //retreat to a safe point
        !| world.GuardPost.HasHealer =>
            .| Vector2.Distance(this.Position,GuardPost.Position)
                > 1.0 =>
                //move towards healer
            .| Vector2.Distance(this.Position,GuardPost.Position)
                <= 1.0 =>
                yield Target,Target.Health,100
        !| _ => //play idle animation
    !| TargetsInRange.Length > 0 =>
        yield TargetsInRange.Head,Target.Health,this.Health
        yield Target,0,this.Health
    !| _ =>
        .&
          //move towards next waypoint
```

```
        .&
          yield Target,Target.Health,this.Health + 1
          wait 1.0

    //...
    //entity constructor

}
```

Below we present the operator syntax in Backus-Naur form and their semantics:

Listing 1: Operators syntax

```
<parallelism> ::= '.&' <expr> {<parallelism>}
<concurrency> ::= '.|' <boolExpr> '=>' <expr> {<concurrency>}
<pre-emption> ::= '!|' <boolExpr> '=>' <expr> {<pre-emption>}
<expr> ::= ...(*
                typical expressions : let , if ,
                for , while , new, query,
                arithmetic expressions, boolean expressions etc.
              *)
                ( wait (<boolExpr | arithExpr>) |
                  <parallelism> | <concurrency> |
                  <pre-emption>) {<expr>}

<arithExpr >  ::= ...(* typical arithmetic expressions *)
<boolExpr >   ::= ...(* typical boolean expressions *)
<literal >    ::= ...(* strings , numbers *)
<queryExpr >  ::= ...(* typical query expressions *)
```

5 Operators optimization

- Static analysis technique to detect dependencies.
- We use these dependencies to generate data structure for scheduling interrupted rules.
  - Each entity maintains a list of entities depending on it, and a list of inactive rules.
  - Maintaining the list according to the dynamics of entities in the game (creation, update, and removal).
- To what extent a game can benefit from such an optimization? Game state changing too fast. Optimization data structures create overhead which surpass the performance gained by our scheduling.
- Dynamic analysis technique to detect the update frequency.
- Rules that update too often will not benefit from our optimization and are not included.
- The choice of the update threshold depends on the data structure we are using for the optimization.
- Further optimization: data structures can be stored in the world instead of locally into entities for better cache coherence.

A problem that arises from the use of some of these operators is that of *busy waiting*. For instance, the healer behaviour requires that he waits for a guard to ask for healing before leaving the guard post. To describe this behaviour we must continuously check if a wounded guard is nearby. The overhead of this operation is minimum for just one guard, but for a greater amount of guards this can seriously affect the game performance, since it is required that the healer checks if any guard is close enough at every game frame. This observation can be generalized for rules which wait for a condition on values of the game state altered by other rules. Consider the following example in Casanova 2:

```
entity E = {
   A : int

   rule A = //rule r1
      wait 1.0
      yield A + 1
   rule A = //rule r2
      wait A % 2 = 0
      //execute rule body b2
   rule A = //rule r3
      wait A % 5 = 0
      //execute rule body b3
   rule A = //rule r4
      wait A % 10 = 0
      //execute rule body b4

   Create() = {A = 0}
}
```

In the snippet of code shown above, rule r_1 increments the field A by 1 every second. Rules r_2, r_3, and r_4 wait until A is multiple of respectively 2, 5, and 10 before executing the rest of their bodies. If we assume that running rule bodies b_2, b_3, and b_4 take 0.5 seconds, and we consider a time span of 12 seconds, then r_2 will be inactive for 10 seconds and run for 2 seconds, r_3 will be inactive for 11 seconds and run for 1 seconds, and r_4 will be inactive for 11.5 seconds and active for 0.5 seconds. This comparison is shown in Figure 2. The time amount in the light grey bars is spent by rules by busy waiting, since in that time interval they keep checking the synchronization condition. We observe that the reactivation of a rule body execution depends only on a change on the fields used in the boolean expression of the `wait` operator, and this change can only be performed by other active rules (in our example only r_1, but in a more general case there could be more than one rule acting on the same field). From this consideration we argue that we can make waiting dependent on the game state rather than on the rule state.

Given that the evaluation of a waiting condition changes only when at least one of the fields affecting it has changed, we keep track of all rules waiting for a change on that field (this can be done statically by examining the expression of a `wait` operator). When a rule evaluates the condition of a `wait` operator, if the condition evaluates to `false`, it is suspended. When another rule changes a field, it resumes any suspended rules with that field. Suspended rules evaluate again the `wait` conditions. Those which evaluate to `false` remain suspended, those which evaluate to `true` become active again and proceed with their execution.

The other operators affected by busy waiting are the concurrency operator and the pre-emption operator (the parallel operator keeps executing its code

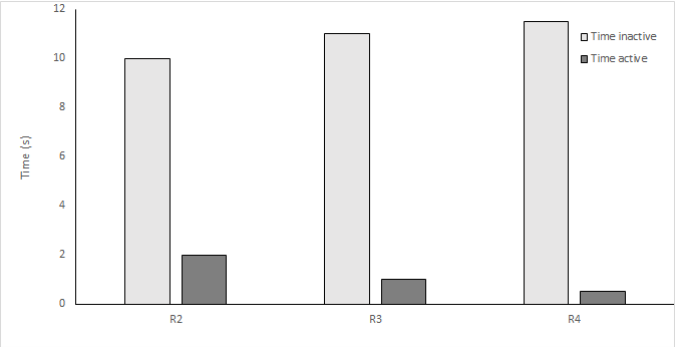

Figure 2: Chart showing the time spent by inactive and active rules

continuously and it does not wait for any condition itself, although its body can contain other operators that do).

6 Evaluation

- Note on the fact we analyse the logic only.
- Compare the time spent by the game loop with and without our optimization technique.
- Introduce the fighting game example (more complex).
- Compare of its execution time with C# ,Python, and unoptimized Casanova 2.0 code.
- Related works:
  - Reactive systems: RXSystem .NET library, Reactive Haskell.
  - Why they are not used for games.
  - Performance comparison.

7 Related works

8 Future works

- Wait scheduling based on time: evaluate only the next-to-happen wait in our time frame.
- Improve the dynamic optimization.
- Hierarchical dependencies in the static optimization.
- Statistical analysis on dependencies.

9 Conlcusions