Casanova: a declarative language for safe games

Giuseppe Maggiore Renzo Orsini
Michele Bugliesi
{maggiore,orsini,bugliesi}@dais.unive.it

Abstract

Games are extremely complex pieces of software which give life to animated virtual worlds. Games require complex algorithms, large worlds filled with many intelligent entities and high-quality graphics, and it all must run in real time.

Building general, high-performance frameworks capable of representing any virtual world has, until now, proven to be an elusive task; existing solutions either sacrifice performance (X3D browsers) or generality (game engines).

In this paper we present a model that formalizes our notion of a correct game. This model gives all the important properties that game developers struggle to maintain by hand, that is the strong normalization of the `tick` function, the independence from the order of the ticks of each entity, and the fact that all active entities of the game state must be ticked exactly once. We define the Casanova language around this model, with the aim of achieving the threefold objective of creating games that are correct, declarative (and thus simpler to write) and fast to run.

We will use a linear type system to enforce the uniqueness of our entities (to ensure exactly one tick for each) and to ensure that ticks do not have overlapping effects. We will impose certain restrictions on the state to avoid infinite loops in the update function (in the shape of cyclic references) and we will use a specialized type system to make sure that all the active coroutines of the game will always yield.

1 Game Model

In this section we will define our notion of a correct game. We observe a series of existing game genres, and we break them down into three fundamental building blocks: entities, rules and behaviors.

Entities represent physical and logical objects; ships, asteroids, projectiles, chairs, timers and effect auras are all entities. Entities follow certain rules, which are verified (run) at every tick of the simulation engine. These rules may be physical (no entities may share the same location) or logical (every item has an active aura that influences the surroundings: a chair makes a place more relaxing for characters who then are less prone to attacking). Rules are synchronous with respect to the game tick. Behaviors represent aspects of the game which are asynchronous to the tick function, for example AIs (which perform single actions that span many ticks) and level activators (which wait for certain conditions to be met before offering access to the next gaming stage).

We now present a simple breakdown of games in terms of entities, rules and behaviors; this list is not omnicomprehensive, but it spans various very different game genres (action games, strategy games, role playing games, arcade games):

- first person shooters:
  - entities are: characters, projectiles, weapons, obstacles, buildings
  - rules are: physics, projectiles damage, picking up weapons, being in cover
  - behaviors are: bot AIs, game logic (capture the flag, other variations), player input
- real time strategy:
  - entities are: units, unit squadrons (if present) buildings
  - rules are: physics, movement (flocking), battle resolution, building queues
  - behaviors are: pathfinding, victory conditions, player input
- Sims:
  - entities are: characters, furniture
  - rules are: furniture auras, social interactions
  - behaviors are: pathfinding, AI, player input
- Puzzle Bobble:
  - entities are: bubbles, bubble queue
  - rules are: bubble movement, bubble blowing, bubble generation
  - behaviors are: level victory, player input

We describe the game G as the pair (E, B) of all the entities and all the behaviors:

$$G = (E, B)$$

Entities are pairs (e, u) of an entity e and its update function u.

We define the interpretation $[\![u]\!]_I$ of an update function as a function from an entity and the game state to its updated version. Since entities update by interacting with each other, the update function cannot take as input the entity e alone. We write:

$$e' = [\![u]\!]_I(e, G)$$

to denote the update of an entity e.

The interpretation $[\![b]\!]_I$ of a behavior b is defined as the function that updates the entire state by executing a single step of a behavior. A step of a behavior is executed by sequentially executing all of its instructions from the previous point of suspension until the next. A behavior suspends itself by invoking the `yield` function.

The interpretation of a set of behaviors $B = b_1, ..., b_m$ is the concatenation of all the interpretations of all the behaviors:

$$[\![B]\!]_I \; = \; [\![b_1]\!]_I \; \circ \; ... \; \circ \; [\![b_n]\!]_I$$

We write:

$$G' = [\![B]\!]_I(G)$$

to denote the application of a step of all behaviors to the game.

The tick function is defined as the function that given a game state $G = (E, B)$ produces the updated game state G' where all entities have been updated according to their rules and all behaviors have been run for one step:

$$\begin{array}{ll} tick(E, B) = G' & \\ \texttt{where} & G' = [\![B]\!]_I((E', \emptyset)) \\ \texttt{and} & E' = \{([\![u]\!]_I(e, G), u) \mid (e, u) \in E\} \end{array}$$

Rules enforce certain properties on each entity; behaviors on the other hand may add, remove or modify any aspect of the game. Behaviors may add or remove new behaviors and entities.

Correctness We define four properties of a correct game; these properties are related to the tick:

1. all rules of each entity are applied exactly once
2. rule application is order-independent
3. tick always terminates

These rules are important because:

1. if a rule is not applied or is applied more than once we risk inconsistent updates, for example an entity which does not move or moves too fast
2. rule application must be order-independent, because all rules represent aspects of the world which happen during one tick; since the tick is the smallest unit of time in a simulation, anything that happens during one tick must behave as if it happened simultaneously

3. tick must always terminate, since otherwise the simulation would hang indefinitely

We now explore a few common mistakes that are commonly performed when coding games by hand. Let us consider a simple game where the entities are asteroids, cannons and projectiles. We define as entities the asteroids, cannons and projectiles. The user clicks a cannon to make it shoot. Rules are:

- asteroids fall towards the bottom of the screen
- projectiles move towards the top of the screen
- asteroids that are hit by a projectile are destroyed
- projectiles that hit an asteroid are removed
- cannons that are hit by an asteroid are destroyed
- the score is incremented by one whenever an asteroid is hit by a projectile
- asteroids that exit the screen are removed

Behaviors are:

- when a cannon is clicked it shoots a projectile
- when the score reaches 1000 the game is won
- when there are no more cannons the game is lost

Let us now consider the mistakes that could be made with respect to the three constraints we have seen above (all rules are executed exactly once, rule application is order-independent and tick is strongly normalizing).

If the first constraint is violated, then an asteroid, projectile or cannon may not be updated correctly or may move or collide more than once; moving more than once during the same tick would multiply an entity velocity by a value greater than one, while colliding more than once would increment the score counter too much.

This constraint could be violated very easily by removing or duplicating the same entity from the entity list.

To see a violation of the second constraint let us consider two possible rules in action; let us assume that rules are executed sequentially and their results are stored in place:

```
module Asteroids =
  (* remove asteroids which are destroyed or which are
      no more visible *)
  asteroids := [a | a <- asteroids, a.Life > 0, a.
     InScreen]
  (* increment the score by the number of destroyed
     asteroids *)
  score := score + [a | a <- asteroids, a.Life <= 0].
     Count * 10

module Projectiles =
  (* increment the score by the number of projectiles
     that hit their target *)
  score := score + [p | p <- projectiles, p.HasHit].
     Count

  (* remove projectiles which hit an asteroid or which
      are no more visible *)
  projectiles := [p | p <- projectiles, p.HasHit =
     false, p.InScreen]
```

If the destroyed asteroids are removed from the state before the score is updated, then the score will never be modified. Projectiles do not suffer from this problem because the score for projectile hits is registered before the hitting projectiles are removed.

This kind of mistake is easy to make and can easily creep in a game, especially as the number of entity types and processing rules grows. Moreover, by splitting the effects on a field (such as `score`) in different modules it may becomes quite hard to fully understand how an entity is processed during a single tick.

Violations of the last constraint are very dangerous, in that they do not produce a logical mistake but rather they would make the game hang, which is essentially the same as an application crash but even more invasive to the user who then needs to manually kill the game process (an operation that is made even harder when the game is running in fullscreen and has claimed full ownership of the graphics card).

2 The Casanova language

In this section we present the Casanova language syntax, typing rules and semantics.

Note on syntactic sugar:

We will use the following syntactic sugar to increase source code readability:

Rather than write:

```
let! _ = b1
in b2
```

we can write:

```
do! b1
in b2
```

Rather than write:

```
let x = t1
in let y = t2
in t3
```

we can write:

```
let x = t1
let y = t2
t3
```

2.1 Simple Example

Before we start, we will give a general idea of how the language works with a small example. We will build a very simple game where asteroids enter the screen from the top, scroll down to the bottom at different velocities and then disappear.

A Casanova program is composed of two portions:

- the state of the game, a series of types arranged hierarchically (typically at least one for the global state and one for the state of each entity); each portion of the game state may contain exactly one rule (a function that computes the new value of the field)
- the main behavior, which performs a series of instructions on all the mutable fields of the state (those marked as `Rule T` or `Var T`; behavior execution is suspended at the `yield` statement, and resumed at the next tick

The state of our simple program is defined as:

```
type Asteroid =
  {
    Y      : Rule float
           :: \(self,y,dt) -> y + dt * self.VelY

    VelY   : float
    X      : float
  }

type GameState =
  {
    Asteroids
        : Rule(Table Asteroid)
        :: \asteroids -> [a | a <- asteroids && a.Y >
           0]

    DestroyedAsteroids
        : Rule<int>
        :: \(state,destroyed_asteroids) ->
           destroyed_asteroids + count([a | a <- !
           state.Asteroids && a.Y <= 0])
  }
```

In a type declaration, the `:` operator means "has type", while the `::` operator means "has rule". Rules can access the game state, the current entity and the time delta between the current and previous ticks.

In the state definition above we can see that the state is comprised by a set of asteroids which are removed when they reach the bottom. Removing these asteroids increments a counter, which is essentially the "score" of our game. Each asteroid moves according to its velocity.

The initial state is then provided:

```
let state0 =
  {
    Asteroids              = []
    DestroyedAsteroids     = 0
  }
```

After defining the state we must give an initial behavior. As can be easily noticed, our game does not generate any asteroids and so the initial state will never change. Since creating asteroids is an activity that certainly must not be performed at every tick (otherwise we could generate in excess of 60 asteroids per second: clearly too many), we need a function that is capable of performing *different* operations on the state depending on time. Since rules perform the *same* operation at every tick, they are unsuited to this kind of processing. Behaviors are built exactly around this need. The behavior for our game is the following:

```
let main state =
  let random = mk_random()
  let rec wait interval =
    {
      let! t0 = time
      do! yield
      let! t = time
      let dt = t - t0
      if dt > interval then
```

```
        return ()
      else
        do! wait (interval - dt)
    }
  let rec behavior() =
    {
      do! wait (random.Next(1,3))
      do! state.Asteroids.Add
          {
            X      = random.Next(-1,+1)
            Y      = 1
            VelY   = random.Next(-0.1,-0.2)
          }
      if state.DestroyedAsteroids < 100 then
        do! behavior()
      else
        return ()
    }
  in behavior()
```

Our behavior declares a random number generator and then starts iterating a function that waits between 1 and 3 seconds and then creates a random asteroid. When the number of destroyed asteroids is greater than 100, the function stops and the game ends (games end when their main behavior terminates).

Notice that behaviors are expressed with two different syntaxes: an ML-style syntax for pure terms (those which read the state and simply perform computations) and an imperative-style syntax for impure terms (those which write the state and interact with time such as wait). The imperative syntax loosely follows the monadic syntax of the F# language, where a monadic block is declared within `{}` parentheses, monadic operations are preceded by either `do!` or `let!` and returning a result is done with the `return` statement.

2.2 Syntax

In the remainder of this section we will adopt the following conventions:

- capitalized items such as `Program` and `StateDef` are grammatical elements
- quoted items such as `'type'` and `'GameState'` are keywords that must appear as indicated
- items surrounded by `[ ]` parentheses such as `[EntityName]` are user-defined strings

The program syntax starts with the definition of the state (a series of type definitions with rules) and is followed by the entry point (the initial state and the initial behavior):

```
Program   ::= StateDef
              Main
StateDef  ::= EntityDefs
```

A type definition is comprised of one of various primitive types such as integers, floating point numbers, two- or three- dimensional vectors, etc. combined into any of the usual composite types known to functional programmers such as tuples, functions, records and sum types. Also, type declarations may contain a rule (which is simply a term, even though with the limitation that only pure functional terms are allowed inside rules). Finally, there are two types for describing behaviors; `UnsafeBehavior T` which represents an unsafe behavior which may never `yield`, and `SafeBehavior T` which represents a safe behavior which never loops indefinitely without `yield`-ing.

```
TypeDef   ::= TypeDef'
            | TypeDef' :: Rule

TypeDef'  ::= '()' | 'int' | 'float' | 'Vector2' | ...
            | TypeDef × TypeDef
            | TypeDef → TypeDef
            | '{' Labels '}'
            | TypeDef + TypeDef
            | Modifier TypeDef
            | 'UnsafeBehavior' TypeDef
            | 'SafeBehavior' TypeDef
            | [EntityName]

Labels    ::= Label; Labels
            | Label

Label     ::= [Name] ':' TypeDef

Rule      ::= Term
```

A `Modifier` for a type definition allows to make a field mutable (`Rule` or `Var`), or to use queries to manipulate that field (`Table`). Also, another important modifier is `Ref` which can be seen as a programmer annotation that tells the compiler how a certain field is just a pointer to another portion of the state and as such it must not be processed recursively:

```
Modifier  ::= Rule
            | Var
            | Table
            | Ref
```

Entities are definied as a series of type definitions with a name which can be referenced anywhere in the state; the last entity to be defined is the game state itself:

```
EntityDefs   ::= ‘type’ ‘GameState’ = TypeDef
               | EntityDef
                 EntityDefs

EntityDef    ::= ‘type’ [EntityName] ‘=’ TypeDef
```

The various entity names are simply replaced with their type definition in the remainder of the program, according to the $[\![\bullet]\!]_{\texttt{MAIN}}$ translation rule:

$$[\![\text{type } \texttt{EntityName} \ = \ \texttt{TypeDef};\ \texttt{EntityDefs}; \texttt{Main}]\!]_{\texttt{MAIN}} =$$

$$[\![\texttt{EntityDefs};\ \texttt{Main}]\!]_{\texttt{MAIN}}[\texttt{EntityName} \mapsto \texttt{TypeDef}]$$

$$[\![\texttt{TypeDef};\ \texttt{Main}]\!]_{\texttt{MAIN}} = \texttt{TypeDef};\ \texttt{Main}$$

The actual type definition of the state may be extracted from the program with the $\sigma(\bullet)$ function, which extracts the type definition and erases all the rules from it; this means that two entities may have the same type with different sets of rules. The function inductively removes all rules from a type declaration:

$$\sigma(\texttt{Entity}; \texttt{EntityDefinitions}) = \sigma(\texttt{EntityDefinitions})$$

$$\sigma(\text{‘}\texttt{typeGameState} =' \texttt{TypeDefinition}) = \sigma(\texttt{TypeDefinition})$$

$$\sigma(\texttt{T} :: \texttt{rule}) = \sigma(T) \qquad \sigma(\texttt{T}_1 \overset{\times}{\underset{\rightarrow}{+}} \texttt{T}_2) = \sigma(\texttt{T}_1) \overset{\times}{\underset{\rightarrow}{+}} \sigma(\texttt{T}_2)$$

$$(...)$$

A term can be a simple, ML-style functional term (we do not give all these possible definitions because they are fairly known) or an imperative behavior. Functional terms can read variables with the `!` operator and can use a Haskell-style table-comprehension syntax:

```
Term         ::= ‘let’ [Var] ‘=’ Term
                 ‘in’ Term
               | ‘letrec’ [Var] ‘=’ Term
                 ‘in’ Term
               | ‘if’ Term ‘then’
                    Term
                 ‘else’
                    Term
               | Term Term
               | !Term
```

```
                | 'add' Term Term
                | ... (* other ML-style terms: fun,
                   types, head, tail for tables, etc. *)
                | [ Term | Predicates ]
                | '{' Behavior '}'

Predicates   ::= ϵ
                | [Var] '<-' Term, Predicates
                | Term, Predicates
```

A behavior defines an imperative coroutine that is capable of reading and writing the state and manipulating time. Behaviors can be freely mixed with terms. The simplest behavior simply returns a result with `return`. The result of a behavior can be plugged inside another behavior with `let!`, which behaves like a monadic binding operator. A variable can be modified inside a behavior with `:=` or `add`; a behavior can suspend itself until the next tick (`yield`) and it may read the current time with `time`.

Behaviors can be combined into more complex behaviors with a small set of combinators. A behavior may spawn another behavior with `run`, be executed in parallel with another behavior with $\vee$ or $\wedge$, be suspended until another behavior completes ($\Rightarrow$), be repeated indefinitely (`repeat`) and be forced to execute in a single tick (`atomic`):

```
Behavior     ::= 'return' Term
                | 'let!' [Var] '=' Term
                  'in' Term
                | Term := Term
                | 'yield'
                | 'time'
                | 'run' Term
                | Term ∨ Term
                | Term ∧ Term
                | Term ⇒ Term
                | 'repeat' Term
                | 'atomic' Term
```

The main program is comprised of two terms: the initial state and the initial behavior:

```
Main    ::= 'let' state0 = Term
            'let' main = Term
```

2.3 Type System

Our language is strongly typed. We will omit some type declarations when obvious, and our language will make use of type inference. Typing rules for ML-style terms are the usual ML-style typing rules; for example:

$$\frac{\Gamma \vdash t_1 : U \quad \Gamma, x : U \vdash t_2 : V}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 \ : V}\ \text{LET} \qquad \frac{\Gamma \vdash c : \mathit{bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \texttt{if } c \texttt{ then } t_1 \texttt{ else } t_2 : T}\ \text{IF}$$

$$\frac{\Gamma \vdash r : \mathit{Var}\ T}{\Gamma \vdash !r : T}\ \text{VAR-GET} \qquad \frac{\Gamma \vdash r : \mathit{Rule}\ T}{\Gamma \vdash !r : T}\ \text{RULE-GET}$$

(...)

Table comprehensions are types thusly:

$$\frac{\texttt{decls}(\Gamma, \texttt{ts}) \vdash t_1 : T}{\Gamma \vdash [\ t_1 \mid t_s\ ]\ \ : \mathit{Table}\ T}\ \text{TABLE} \qquad \texttt{decls}(\Gamma, \epsilon) = \Gamma$$

$$\frac{\Gamma \vdash t : \mathit{Table}\ T}{\texttt{decls}(\Gamma, (\texttt{x} \leftarrow \texttt{t}, \texttt{ts})) = \texttt{decls}((\Gamma, \texttt{x} : \texttt{T}), \texttt{ts})}$$

$$\frac{\Gamma \vdash t : \mathit{bool}}{\texttt{decls}(\Gamma, (\texttt{t}, \texttt{ts})) = \texttt{decls}(\Gamma, \texttt{ts})}$$

(...)

Linearity Our type system needs to ensure that updating each entity of the state is a safe operation, that is the same entity will be updated *exactly* once. The semantics function that we give further in this Section guarantees that all entities are updated *at least* once, but duplicate entities may be updated twice or more. Updating an entity more than once is dangerous because it may lead to unexpected behaviors, but there is another downside to duplicates: with duplicates, rules are no more order-independent. With duplicates, the same entity may be subject to more than one rule, and thus the same entity may be modified twice in (possibly) irreconcilable ways.

We now show a few examples of how we may produce these situations.

A common error is duplication of a field either inside a behavior or at initialization time; for example, given a record type `T` with two fields `Position:Rule U` and `Velocity:Rule U` we may erroneously write:

```
let r:Rule U = mk_cell ...

let (x:T) =
  {
    Position = r
    Velocity = r
  }
```

Another common error is adding to a table an element from the table itself; for example, given a variable `t` of type `Rule Table T` we may write:

```
t.add (t.head)
```

The final, common error we show is duplicating data with two symmetric rules:

```
Asteroids₁
   : Rule(Table Asteroid)
   :: fun self -> [a | a <- self.Asteroids₁ ∪ self.
      Asteroids₂, p₁ a]

Asteroids₂
   : Rule(Table Asteroid)
   :: fun self -> [a | a <- self.Asteroids₁ ∪ self.
      Asteroids₂, p₂ a]
```

unless $p_1 = \neg p_2$ there will be duplicates between $\texttt{Asteroids}_1$ and $\texttt{Asteroids}_2$.

To solve this problem we use two techniques; first, we make `Rule` a *linear* type. Second, we give rules read-only (`Ref`) access to the state, so that copying an external portion of the state into another field will require a deep cloning of that portion of the state and not just variable sharing.

We do not discuss all the typing rules required to enforce; instead we show some of the most significant:

$$\frac{\Gamma \vdash t : \texttt{Ref}\ \{l_1 : T_1; ...l_n : T_n\}}{\Gamma \vdash t.l_i\ :\ \texttt{Ref}\ T_i} \quad \text{FOREIGN-LOOKUP}$$

$$\frac{\begin{array}{cc}\text{Linear}(U) & \Gamma \vdash t_1 : U\end{array} \\ (\Gamma \setminus \text{LFV}(t_1)), x : U \vdash t_2 : V}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2\ :\ V} \quad \text{LINEAR-LET} \qquad \texttt{where}$$

$$\text{Linear}(T) = (\exists \alpha \mid \texttt{Rule}\ \alpha\ \in \text{range}(T)) \qquad \text{NonLinear}(T) = \neg\, \text{Linear}(T)$$

$$\text{LFV}(t) = \{(v : \alpha) | (v : \alpha) \in \text{FV}(t)\ \wedge \text{Linear}(\alpha)\}$$

Behaviors We also state another informal restriction, that is function types may not have rules; so, a type such as $(U :: \mathit{Rule}) \rightarrow V$ is forbidden and generates a compile-time error.

The first typing rules for behaviors are the monadic typing rules which allow to build and consume basic behaviors:

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash \texttt{return}\ x :\ \mathit{UnsafeBehavior}\ T}\ \text{RETURN}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : b_1\ U \\ \Gamma, x : U \vdash t_2 : b_2\ V\end{array}}{\Gamma \vdash \texttt{let!}\ x = t_1\ \texttt{in}\ t_2 : b_1 \sqcap b_2\ V}\ \text{BIND} \qquad \texttt{where}$$

$$\mathit{SafeBehavior} \sqcap \_ = \mathit{SafeBehavior} \qquad \_ \sqcap \mathit{SafeBehavior} = \mathit{SafeBehavior}$$

$$\mathit{UnsafeBehavior} \sqcap \mathit{UnsafeBehavior} = \mathit{UnsafeBehavior}$$

Behaviors may also be suspended for a tick (to wait for an application of all rules or to synchronize between behaviors, for example), they may read the current time (in fractional seconds) or they may spawn other behaviors:

$$\frac{}{\texttt{yield} : \mathit{SafeBehavior}\ ()}\ \text{YIELD}$$

$$\frac{}{\texttt{time} : \mathit{UnsafeBehavior}\ \mathit{float}}\ \text{TIME}$$

$$\frac{\Gamma \vdash t : \mathit{SafeBehavior}\ ()}{\Gamma \vdash \texttt{run}\ t : \mathit{UnsafeBehavior}\ ()}\ \text{RUN}$$

Behaviors are the only places where unrestricted modification of the state may happen; behaviors may indiscriminately write any portion of the state:

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Var}\ (\mathit{Table}\ U) \\ \Gamma \vdash t_2 : U\end{array}}{\Gamma \vdash \texttt{add}\ t_1\ t_2 : \mathit{UnsafeBehavior}\ ()}\ \text{VAR-TABLE-ADD}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Rule}\ (\mathit{Table}\ U) \\ \Gamma \vdash t_2 : U\end{array}}{\Gamma \vdash \texttt{add}\ t_1\ t_2 : \mathit{UnsafeBehavior}\ ()}\ \text{RULE-TABLE-ADD}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Var}\ U \\ \Gamma \vdash t_2 : U\end{array}}{\Gamma \vdash t_1{:=}t_2 : \mathit{UnsafeBehavior}\ ()}\ \text{VAR-SET}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathit{Rule}\ U \\ \Gamma \vdash t_2 : U\end{array}}{\Gamma \vdash t_1{:=}t_2 : \mathit{UnsafeBehavior}\ ()}\ \text{RULE-SET}$$

Conditionals on behaviors propagate the least safe of the two behavior types:

$$\frac{\Gamma \vdash c : bool \quad \Gamma \vdash t_1 : b_1\ U \quad \Gamma \vdash t_2\ :\ b_2\ U}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t_1\ \texttt{else}\ t_2\ :\ b_1 \sqcup b_2 U}\ \text{IF-BEHAVIOR} \qquad \texttt{where}$$

$$UnsafeBehavior \sqcup \_ = UnsafeBehavior$$

$$\_ \sqcup UnsafeBehavior = UnsafeBehavior$$

$$SafeBehavior \sqcup SafeBehavior = SafeBehavior$$

We also support a small behavior calculus. Two behaviors may be executed concurrently (the first one that terminates returns its result while the other behavior is discarded), or they may be executed in parallel (when both terminate their results are returned together). A behavior may also act as a guard ($\Rightarrow$) for another behavior, that is until the first behavior does not terminate with a result the second behavior is kept waiting. Finally, a behavior may be repeated indefinitely or it may be forced to run inside a single tick:

$$\frac{\Gamma \vdash t_1 : b_1\ U \quad \Gamma \vdash t_2 : b_2\ V}{\Gamma t_1 \vee t_2 : b_1 \sqcup b_2\ (U + V)}\ \text{CONCURRENT}$$

$$\frac{\Gamma \vdash t_1 : b_1\ U \quad \Gamma \vdash t_2 : b_2\ V}{\Gamma t_1 \wedge t_2 : b_1 \sqcup b_2\ (U \times V)}\ \text{PARALLEL}$$

$$\frac{\Gamma \vdash t_1 : SafeBehavior\ (U + ()) \quad \Gamma \vdash t_2 : U \rightarrow b\ V}{\Gamma t_1 \Rightarrow t_2 : b\ V}\ \text{GUARD}$$

$$\frac{\Gamma \vdash t : SafeBehavior\ ()}{\Gamma \vdash \texttt{repeat}\ t : SafeBehavior\ ()}\ \text{REPEAT}$$

$$\frac{\Gamma \vdash t :\ b()}{\Gamma \vdash \texttt{atomic}\ t : UnsafeBehavior\ ()}\ \text{ATOMIC}$$

The typing rules on behaviors force them to be written in such a way as to guarantee statically that no behavior will run indefinitely without yielding. This is seen easily because the only two operators capable of looping are `repeat` and $\Rightarrow$, and both require a `Safebehavior` as input.

We now show a small list of dangerous behaviors that are forbidden by our type system:

```
repeat { r := v }

repeat {if cond then r := v else yield}

{ return None } => fun () -> yield
```

The inability to eliminate a behavior unless inside another behavior is important because it allows us to force rules to not contain behaviors; thanks to this limitation we can ensure that the execution of rules may only read from the state and never write to it, and so rules can be made to behave as if they are executed simultaneously without risking complex interdependencies. This simplifies many instances of game programming; for example, consider the rules seen in the example at the beginning of the section:

```
Asteroids
     : Rule(Table Asteroid)
     :: \asteroids -> [a | a <- asteroids && a.Y > 0]

DestroyedAsteroids
     : Rule<int>
     :: \(state,destroyed_asteroids) ->
         destroyed_asteroids + count([a | a <- !state.
         Asteroids && a.Y <= 0])
```

If rules are executed sequentially from top to bottom, then when an asteroid is eliminated that same asteroid will not be available anymore when computing the number of asteroids waiting for deletion.

2.4 Accepted Programs

To determine if a game is correct or not we require for it to pass a static analyisis which is comprised of two steps: its state definition must be well-formed (the wf(•) function) and its `main` must be typed correctly.

The wf(•) function requires that a state definition's rules are typed correctly:

$$\mathrm{wf}(\texttt{type[EntityName] = TypeDef; EntityDefinitions}) =$$

$$\mathrm{wf}(\texttt{TypeDef}, \texttt{EntityName}) \wedge \mathrm{wf}(\texttt{EntityDefinitions})$$

$$\mathrm{wf}(\texttt{Primitive}, \texttt{EntityName}) = \texttt{true}$$

$$\mathrm{wf}(\texttt{CompositeType(Types)}, \texttt{EntityName}) = \bigwedge_{\texttt{T} \in \texttt{Types}} (\mathrm{wf}(\texttt{T}, \texttt{EntityName}))$$

$$\mathrm{wf}(\texttt{Rule T :: rule}) = \mathrm{wf}(\texttt{T}, \texttt{EntityName}) \wedge$$

$$\vdash \texttt{rule} \ : rule - fun(T, EntityName)$$

$$\mathrm{wf}(\texttt{Var T}, \texttt{EntityName}) = \mathrm{wf}(\texttt{T}, \texttt{EntityName})$$

$$\mathrm{wf}(\texttt{Table T}, \texttt{EntityName}) = \mathrm{wf}(\texttt{T}, \texttt{EntityName})$$

$$\mathrm{wf}(\texttt{Foreign T}, \texttt{EntityName}) = \texttt{true}$$

$$\texttt{where} \qquad \text{rule-fun}(T, \texttt{EntityName}) =$$

$$\texttt{Foreign(GameState)} \times \texttt{Foreign(EntityName)} \times \texttt{T} \times \texttt{float} \rightarrow \texttt{T}$$

Given a game source:

```
StateDefinition

let state0 = t₁
let main   = t₂
```

its compilation succeeds if the following is satisfacted:

$$\mathrm{wf}(\texttt{StateDefinition}) \wedge$$

$$\vdash t_1 : \sigma(\texttt{StateDefinition}) \wedge$$

$$\vdash t_2 : \sigma(\texttt{StateDefinition}) \rightarrow \texttt{SafeBehavior()}$$

2.5 Semantics

We now define the semantics function of a Casanova program.

We start by defining our memory model. Our memory is defined as a map from rules and variables into values:

```
m = ϵ
  | m[r → v]
  | m[r ⇒ v]
```

`m[r → v]` means that `r`, which has type `Var T` or `Rule T` has value `v`, while `m[r ⇒ v]` means that `r`, which has type `Rule T`, has a pending assignment of value `v`. The execution of rules does not modify assignments of the form `m[r → v]`, and it will only add assignments of the form `m[r ⇒ v]`. After all rules have been executed, then we use the compacting function ⊕:

```
⊕(ϵ) = ϵ
⊕(m[r → v]) = (⊕m)[r → v]
⊕(m[r ⇒ v]) = (⊕m)[r → v]
```

The `!` operator (valid on both rules and variables) is defined as:

```
!r (m[r’ → v]) = if r = r’ then v else !r m
!r (m[r’ ⇒ v]) = !r m
```

variables may never be null, since there is no way of declaring a variable without initializing it at the same time; this way we are assured that variable lookups will always succeed in finding a value.

The `:=` operator (valid on both rules and variables) is defined as:

```
(r := v) m = m[r → v]
```

Notice that `:=` cannot be used inside the body of a rule, thanks to the limitation that the type system imposes that unrestricted assignments can only happen inside behaviors.

We can now define the `update_rules` function, which builds the program that will run all the rules; this program takes as input the memory and after each imperative statement it returns the modified memory. The `update_rules` function inductively processes the state and applies each rule it finds. The function does not recursively process those portions of the state with type `Foreign T` for some `T`.

The `update_rules` function traverses all entity definitions starting from the beginning of the program; it also invokes the compacting function ⊕ to apply all the pending changes caused by all the rule executions:

```
update_rules (StateDefinition; Main) m =
  ⊕(update_state StateDefinition m)

update_state (type EntityName = TypeDef;
   EntityDefinitions) =
  entities_update[(update_state EntityName EntityName)
      ↦ entity_update]
  where
    entities_update = update_state EntityDefinitions
  and
    entity_update = update_entity TypeDef TypeDef

update_state (type GameState = TypeDef) =
  update_entity TypeDef TypeDef
```

The update_state function inductively processes type definitions; on each type definition it goes in search for rules and applies those rules with the update_entity function:

```
update_entity E T m (s,(e:E),(t:T),dt) = m (* when T
    is a primitive type such as (), int, ... *)

update_entity E (U → V) m (s,e,f,dt) = m

update_entity E (SafeBehavior T) m (s,e,f,dt) = m

update_entity E (UnsafeBehavior T) m (s,e,f,dt) = m

update_entity E (Foreign T) m (s,e,f,dt) = m

update_entity E (U × V) m (s,e,(u,v),dt) =
  update_entity E U (update_entity E V m (s,e,v,dt)) (
     s,e,u,dt)

update_entity E (U + V) m (s,e,(Left u),dt) =
   update_entity E U m (s,e,u,dt)

update_entity E (U + V) m (s,e,(Right v),dt) =
   update_entity E V m (s,e,v,dt)

update_entity E ({l1:T1,...,ln:Tn}) m (s,e,r,dt) =
  update_entity E T1 (... (update_entity E T1 m (s,e,r
     .ln,dt)) ... ) (s,e,r.l1,dt)

update_entity E ([EntityName]) m (s,e,e',dt) = (
   update_entity [EntityName] [EntityName]) m (s,e',e
   ',dt)

update_entity E (Var T) m (s,e,r,dt) = update_entity E
    T m (s,e,!r m,dt)

update_entity E (Table T) m (s,e,t,dt) = [
   update_entity E T m (s,e,x,dt) | x ← t]

update_entity E (Rule T :: rule) m (s,e,r,dt) =
  update_entity E T (m[r ⇒ (⟦rule⟧_I m (s,e,!r m,dt)]) (
     s,e,!r m,dt)
```

Where $[\![\bullet]\!]_I$ m is the well-known semantics of the pure lambda calculus, augmented with the rule that:

$[\![!r]\!]_I$ m = !r m.

Behavior semantics is simpler than rule semantics. We update a list of behaviors, where each behavior may modify the memory or run new behaviors. Each behavior may also perform regular functional computations which are processed according to $[\![\bullet]\!]_I$m.

We define the `update_behaviors` function which folds the `update_behavior` over all behaviors; each single behavior update modifies the memory and returns a set of behaviors to run at the next update:

```
update_behaviors ({b1,...,bn},m) =
  ((b1',bs'),m'')
  where
    (),b1',m' = update_behavior m b1
  and
    bs',m'' = update_behaviors ({b2,...,bn},m')
```

The `update_behavior` function executes a behavior until the next `yield`, immediately writing assignments to the memory; we omit the definition of this function for regular pure functional terms, and instead give it only for behavior terms:

```
update_behavior m (return v) = v,{},m

update_behavior m (let! _ = yield in t) = (),(t),m

update_behavior m (let! _ = (run b) in t) =
  v,(b,b'),m'
  where
    v,b',m' = update_behavior m (t)

update_behavior m (let! _ = (r := v) in t) =
  v,b,m'
  where
    v,b,m' = update_behavior (m[r → v]) (t)

update_behavior m (let! x = !r in t) =
  v,b,m'
  where
    v,b,m' = update_behavior ((λx . t) (!r m)) m

update_behavior m (let! x = !r in t) =
  v,b,m'
  where
    v,b,m' = update_behavior ((λx . t) (!r m)) m

update_behavior m (let! x = t1 in t2) =
  update_behavior ((λx . t2) t1) m
```

The two update functions, one for rules and one for behaviors are combined into the final update function which takes as input the game state, the set of current behaviors and the current memory and which returns the updated behaviors and memory. The state never changes, but the memory it points to may:

```
update Program state bs m =
  bs',m''
  where
    bs',m' = update_behaviors (bs,m)
  and
    m'' = update_rules Program m' state
```

2.6 Correctness

We have stated in Sec. 1 that a correct game according to our model respects the following rules:

1. all rules of each entity are applied exactly once
2. rule application is order-independent
3. tick always terminates

We will now briefly discuss why our system forces games to respect these rules:

1. is respected because the semantics function explores the entire state recursively and applies its rules making sure that each rule is applied at least once; the linearity of the `Rule` type makes sure that the same cell is never processed more than once with one or more rules
2. is respected because rule application cannot access the result of already completed rules because those results are stored in memory as $m[r \Rightarrow v]$ but each rule may only read memory cells of the form $m[r \rightarrow v]$; also, the linearity of the `Rule` type ensures that no two rules write the same cell with $m[r \Rightarrow v]$, because all r's are distinct
3. is respected because the state is not cyclic (apart from `Foreign` declarations, which are not processed recursively) and because only behaviors with type `YieldBehavior` can be run, that is accepted behaviors never run indefinitely without yielding

3 Compilation

In this section we will show an outline of the Casanova compiler. In Fig. 1 we can see that compilation is a four step process.

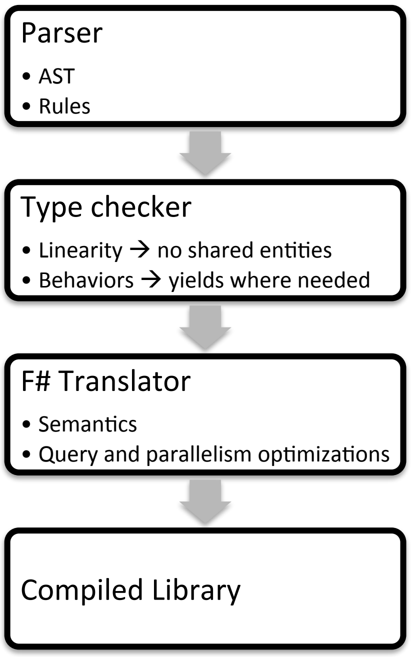
Parser
• AST
• Rules
Type checker
• Linearity → no shared entities
• Behaviors → yields where needed
F# Translator
• Semantics
• Query and parallelism optimizations
Compiled Library

Figure 1: Compilation process

In the first step we parse the program source code. The output is an AST which represents the program, with the rules and types as annotations.

The AST is fed to the type checker, which makes sure that the program does not contain incorrect operations. It makes sure that no duplication of `Rule T` values happens, which would cause inconsistencies when applying rules. The type checker also makes sure that behaviors are typed correctly, thereby ensuring that all behaviors are typed as `Safebehavior` and thus they do not cycle indefinitely without invoking `yield`.

The resulting AST is then processed by the F# translator, which generates F# code that defines the state and entity types and which implements the semantic function.

The F# translator also performs some optimizations which we will discuss in the next paragraph.

Finally, the resulting F# program is compiled by the standard F# compiler thereby generating a library which can then be used as the game engine for an XNA (Windows PCs, Xbox 360 or Windows Phone 7) or MonoTouch (iPad, iPhone or Android) game.

Optimization Casanova performs three main optimizations.

The first optimization is a very simple one: memory recycling; even if simple, it can prove very effective in all those platforms (such as the Xbox 360) with a slow garbage collector. Memory recycling means that `Rule T` fields allocate a double buffer for storing both the $\texttt{m}[r \rightarrow v]$ value and the $\texttt{m}[r \Rightarrow v]$ value. Applying the $\oplus\texttt{m}$ operator simply requires swapping the two buffers. The `Rule T` datatype is defined as:

```
type Rule<'a> =
  {
    Values              : 'a[]
    FrameIndex          : int ref
  }
  member private this.ValueIndex
    with get() = this.FrameIndex.Value % this.
        NumValues
  member private this.ValueIndex'
    with get() = (this.FrameIndex.Value + 1) % this.
        NumValues
  member this.Value
    with get() = this.Values.[this.ValueIndex]
    and set v' = this.Values.[this.ValueIndex] <- v'
  member this.Value'
    with get() = this.Values.[this.ValueIndex']
    and set v' = this.Values.[this.ValueIndex'] <- v'
```

all `Rule T`'s share the same reference to the current frame index. Whenever we wish to swap the references (that is when we apply the $\oplus$ function) then we just increment the `FrameIndex` without any need for traversing the entire state to manually set all `Rule T`'s.

This optimization can be extended to tables: at the beginning of each update, all values of type `Rule (Table T)` get their `Value'` cleared; clearing a table does not deallocate its elements: rather, it simply sets the counter of elements to zero, while keeping the previous memory allocated.

This strategy helps reducing the amount of garbage collection needed, sometimes giving large speedups as we can see in Section 5.

The second optimization takes advantage of the static constraint that rules are linear: this means that no rules write the same memory location. We also know that rules may not freely write any references. These two facts guarantee thread safety, that is we may run or rules in parallel. Casanova dynamically allocates twice as many threads as the number of cores of the machine. Threads are only used to process lists of entities in the `GameState`, but no further multi-threading is performed: if an entity should contain many sub-entities those will all be processed sequentially in the same thread. This is needed to avoid creating too many threads; an excessive number of threads may even cause so much overhead that the benefits of parallelization are inferior to the losses in performance caused by the cost of threads.

The i^{th} thread of n will process the i^{th} portion of each top-level table of the game state. This means that if the state is defined as:

```
type GameState =
  {
    Asteroids   : Table Asteroid
    Projectiles : Table Projectile
  }
```

then the i^{th} thread will run the function:

```
let thread state n i =
  for j = i * state.Asteroids.Count / n
      to (i+1) * state.Asteroids.Count / n do
    update state.Asteroids.[j]
  for j = i * state.Projectiles.Count / n
      to (i+1) * state.Projectiles.Count / n do
    update state.Projectiles.[j]
```

Unless the number of entities is very small or behaviors are very complex and computationally intensive, then the gains obtained by parallelization can be very high; the best results may even divide the duration of a tick by the number of threads, even if this is rarely the case.

The final optimization is query optimization. Nested list comprehensions (also known as "joins" in the field of databases) can have high computational costs; for example, the query:

```
type Asteroid =
  {
    CollidingProjectiles
      : Rule(Table(Foreign(Projectile)))
      :: \(state,self) -> [p | p <- state.Projectiles,
           collides(self,p)]
  }
```

has a computational complexity of $O(n_p \times n_a) = O(n^2)$, where n_p is the number of projectiles, n_a is the number of asteroids and n is the maximum between the two. Such a complexity is unacceptable when we start having a large number of asteroids and projectiles, because it may severely limit the maximum number of entities supported by the game.

We use the same physical optimization techniques used in modern databases: we build an index to speed up our collision detection. In particular, we observe that most asteroids and projectiles are so far away that testing them for collision does not make sense. We partition the space of the playing area into various blocks and we assign all our projectiles to the blocks they belong to; this operation costs $O(n_p)$ if blocks are a uniform grid and $O(n_p \log n_p)$ if blocks are of variable size and represented with a tree. For each asteroid, we find the blocks it belongs to ($O(n_a)$ or $O(n_a \log n_a)$) and then check for collisions only with the projectiles in those blocks. The final cost of the operation is $O(n)$ for hash maps and $O(n \log n)$ for trees.

An example hash map optimization could be the following. We add the following declaration to the game state:

```
type GameState =
  {
    ...
    Blocks   : Block[][]
  }
```

where a `Block` contains a list of projectiles.

In the update function we start by clearing the `Blocks` index and we fill it again with the updated projectiles:

```
let update_state (state:GameState) (dt:float32) =
  for b in state.Blocks do
    b.Clear()

  for p in state.Projectiles do
    for b in p.Blocks do
      b.Add p
```

Benchmarks show that the costs of rebuilding the index are similar to modifying it, especially for trees. In this sense we confirm a similar result found in.

Collision detection will now become:

```
let update_asteroid (state:GameState)
                    (self:Asteroid)
                    (dt:float32) =
  for b in self.Blocks do
    for p in b.Projectiles do
      if collides self p then
        self.CollidingProjectiles.Add p
```

A small bottleneck of this computation is the clearing phase, because it forces us to iterate all the blocks (which may be a large number for increased optimization) even if most of those are empty. For this reason, we further augment the state to track those blocks that contain projectiles (and thus which need clearing):

```
type GameState =
  {
    ...
    NonEmptyBlocks  : Table (BlockIndex)
  }
```

Now the update function becomes:

```
let update_state (state:GameState) (dt:float32) =
  for bi in state.NonEmptyBlocks do
    state.Blocks.[bi].Clear()

  for p in state.Projectiles do
    for b in p.Blocks do
      b.Add p
      state.NonEmptyBlocks.Add b.Index
```

We must stress the importance of this last optimization. By having a function of quadratic complexity in the number of entities we are forcing our game to run with a maximum number of entities. Less entities often make for a less compelling game, because the world is less complex and the challenges are smaller. This class of optimizations allows the game performance to be less dependent on the number of entities. This means that we may design grander worlds with thousands of units and *with no additional development complexity*.

4 Case Studies

In this section we show the implementation of the XNA Spacewar starter kit in Casanova.

The state definition requires defining the game state and the game entities; the entities are the two players (who both are represented with ships), a series of asteroids, a series of projectiles and the sun in the middle of the playing field.

Each entity contains (or is outright) a value of type `Entity`. `Entity` contains a position, a velocity and performs collision detection with the other entities of the state. If a collision happens, then the entity's life is decreased. When an entity has life smaller or equal to zero, then it is removed. Projectiles are an exception in that they are removed when their age exceeds 3 seconds, even if no collisions take place.

The state definition, together with its rules, is the following:

```
type GameState =
  {
    Player1 : Player
    Player2 : Player

    Ships
      : Rule<Table<Ref<Ship>>>
      :: fun state -> [state.Player1.Ship; state.
         Player2.Ship]
    Asteroids
      : Rule<Table<Asteroid>>
      :: fun state -> [a | a <- state.Asteroids, a.
         Life > 0.0f]
    Projectiles (* table of projectiles with life > 0
        and age < 3 *)
    Sun     : Star

    GameStatus
      : Rule<GameStatus>
      (* check if a ship has life < 0, and assign
          winner;
          in case of tie assign GameOver(None) *)
  }

type GameStatus = Playing | GameOver of Option<Player>
```

A generic entity is then defined as:

```
type Entity =
  {
    Position
      : Rule<Vector2>
      (* increment by velocity * dt, unless a
          collision is happening *)
    PrevPosition
      : Rule<Vector2>
      (* store position, to backtrack in case of
          collision *)
    Velocity
      : Rule<Vector2>
      (* invert in case of collision, otherwise
          accelerate towards sun *)

    Radius : float32
    Life
      : Rule<float32>
      :: fun self -> !self.Life - !self.
          ColliderAsteroids.Count * 10.0f
                             - !self.
                                ColliderProjectiles.
                                Count * 5.0f
                             - !self.ColliderShips.
                                Count * 10.0f
                             - !self.ColliderSun.Count
                                 * 100.0f

    ColliderAsteroids
      : Rule<Table<Asteroid>>
      :: fun (state,self) -> [a | a <- state.Asteroids
```

```
         , collides a self]

    ColliderShips
      : Rule<Table<Ship>>
      :: fun (state,self) -> [s | s <- state.Ships,
         collides s self]

    ColliderProjectiles
      : Rule<Table<Projectile>>
      :: fun (state,self) -> [p | p <- state.
         Projectiles, p.Shooter <> self, collides p.
         Entity self]

    ColliderSun
      : Rule<Table<Star>>
      :: fun (state,self) -> if collides state.Sun
         self then [state.Sun] else []

    Colliding
      : Rule<bool>
      :: fun self -> !self.ColliderAsteroids.Count + !
         self.ColliderShips.Count + ... > 0
  }
```

The concrete game entities are:

```
type Ship = Entity
type Asteroid = Entity
type Projectile =
  {
    Entity      : Entity
    Shooter     : Ref<Entity>
      : Rule<float32>
      :: fun (self,dt) -> !self.Age + dt
  }
type Star =
  {
    Position : Vector2
    Radius   : float32
  }
type Player =
  {
    Ship  : Ship
    Score
      : Rule<int>
      :: fun (state,self) -> !self.Score + !self.Hits.
```

```
        Count

    Hits
      : Rule<Table<Projectile>>
      :: fun (state,self) -> [p | p <- state.
         Projectiles, p.Shooter =    }
```

The initial state sets up the two players, a set of 12 asteroids, the sun and the current gameplay status (which indicates whether the game is running or is over). We simplify initialization of an entity with the `mk_entity` function, which initializes the various collider fields to an appropriate default value. The initial state is defined as:

```
let state0 =
  let mk_entity position velocity size life = (* ...
     *)

  let player1 =
    {
      Ship           = mk_entity (Vector2(200.0f, 240.0
         f)) (Vector2.UnitY * 50.0f) 10.0f 100.0f
      Score          = mk_cell 0
      Hits           = mk_cell Table(50)
    }
  let player2 =
    {
      Ship           = mk_entity (Vector2(600.0f, 240.0
         f)) (-Vector2.UnitY * 20.0f) 10.0f 100.0f
      Score          = mk_cell 0
      Hits           = mk_cell Table(50)
    }

  let asteroids() =
    Table(
      seq{
        for i = 1 to 12 do
          let x = (float32 i) * 70.0f + 10.0f
          let a = mk_entity (Vector2(x,480.0f)) (rand.
             NextVector2 * 50.0f) 10.0f 100.0f
          yield a
      })

  let state =
    {
      Player1              = player1
      Player2              = player2
      Ships                 = mk_cell Table(2)
      Asteroids             = mk_cell asteroids
      Projectiles           = mk_cell Table(1000)
      Sun                   =
        {
          Position = Vector2(400.0f,240.0f)
          Radius   = 20.0f
        }

      GameStatus            = mk_cell Playing
    }
  state
```

The main behavior of the program is defined as a loop that iterates until the game state is `Playing`. While that is not the case, the loop continuously instantiates projectiles and asteroids according to user input and to maintain the initial number of asteroids even when existing ones are destroyed. The main behavior is defined as:

```
let behavior0 (state:GameState) : Behavior<Unit,Unit>
   =
  let rec behavior() =
  {
    do! begin_
    if state.Projectiles.Value.Count < 200 then
      let shooter = state.Player1.Ship
      do state.Projectiles.Value.Add (* create new
          projectile *)
      let shooter = state.Player2.Ship
      do state.Projectiles.Value.Add (* create new
          projectile *)
      do! wait 0.1f
    if state.Asteroids.Value.Count < 12 then
      do state.Asteroids.Value.Add (* create new
          random projectile *)
      do! wait 0.05f
    do! yield_
    if state.GameStatus = Playing then
      do! behavior()
  }
```

5 Benchmarks

We have slightly modified the original sample so that testing could be automated. For this reason we have removed the 30 seconds time limit of each level, we have removed the victory and ending conditions, we have automated ships movement and shooting and we have increased the maximum number of asteroids and projectiles to 12 and 200 respectively. This way we have obtained an automated stress test.

We have also removed all rendering features, to avoid benchmarking rendering algorithms: Casanova does not generate rendering code, so such a comparison would have been meaningless.

We have benhcmarked the modified sample on both the Xbox 360 and a 1.86 Ghz Intel Core 2 Duo with an nVidia GeForce 320M GPU and 4GB of RAM. The resulting framerates of the original sample are reported in Table 1:

Platform	Framerate
Xbox	9
PC	8

Table 1: Spacewar Sample Framerate

The Casanova program has been compiled with all possible combinations of query optimization and multi-threaded optimization. The resulting framerates of the rewritten sample run under the Xbox 360 are reported in Table 2:

Query Optimization	MT Optimization	Framerate
No	No	3
No	Yes	6
Yes	No	18
Yes	Yes	22

Table 2: Casanova Xbox Framerate

The resulting framerates of the rewritten sample run under a PC are reported in Table 3:

Query Optimization	MT Optimization	Framerate	% of CPU used
No	No	22	50
No	Yes	93	97
Yes	No	510	50
Yes	Yes	577	85

Table 3: Casanova PC Framerate

As we can see, full Casanova optimization always beats the original source by at least a factor of 2. The Xbox implementation suffers from the generation of garbage, which is a known problem of the XNA implementation on the console

([11]); indeed, profiling the garbage collector shows that large amounts of temporary memory are being generated by the program. It is noticeable that full optimization on the PC actually increased performance by almost two orders of magnitude: such an impressive increase was quite unexpected.

As a final remark, it is worth noticing that while the original sample includes more than one thousand lines of code the length of the corresponding Casanova program is 348 lines long. The Casanova source easily fits a few pages, while navigating the original source may prove a bit complex because of its sheer size.

References

[1] Casanova project page. casanova.codeplex.com/.

[2] Entertainment software association. http://www.theesa.com.

[3] Evolve your hierarchy. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy.

[4] Games using lua as a scripting language. http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games.

[5] Haskell list comprehensions. http://www.haskell.org/haskellwiki/List_comprehension.

[6] Inheritance vs aggregation in game objects. http://gamearchitect.net/Articles/GameObjects1.html.

[7] Irrlicht engine. http://irrlicht.sourceforge.net.

[8] Ogre engine. http://www.ogre3d.org.

[9] Safety and performance in x3d: Technical report. ???

[10] Scripting in unity. http://unity3d.com/support/ documentation/ScriptReference/index.Coroutines_26_Yield.html.

[11] Slow garbage collection on the xbox. http://blogs.msdn.com/b/shawnhar/archive/2007/06/29/how-to-tell-if-your-xbox-garbage-collection-is-too-slow.aspx.

[12] Unreal engine. http://www.unrealengine.com/.

[13] Unrealscript documentation. http://unreal.epicgames.com/UnrealScript.htm.

[14] The xna framework. http://msdn.microsoft.com/xna.

[15] Xna spacewar 4. http://create.msdn.com/education/catalog/sample/spacewar.

[16] Scott Bilas. http://scottbilas.com/files/2002/gdc_san_jose/game_objects_paper.html.

[17] Mat Buckland. *Programming Game AI by Example.* Jones & Bartlett Publishers, 1 edition, September 2004.

[18] Leonard Daly and Don Brutzman. X3d: extensible 3d graphics standard. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 22:1–22:6, New York, NY, USA, 2008. ACM.

[19] Bruce Dawson. Game scripting in python. http:// www.gamasutra.com/features/ 20020821/dawson_ pfv.htm, 2002. Game Developers Conference Proceedings.

[20] L. H. de Figueiredo, W. Celes, and R. Ierusalimschy. Programming advanced control mechanisms with lua coroutines. In *Game Programming Gems 6*, pages 357–369, 2006.

[21] Ana L. de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.

[22] Mark. Deloura. *Game Programming Gems.* Charles River Media, Inc., Rockland, MA, USA, 2000.

[23] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32:263–273, August 1997.

[24] Michael Haller Fh, Michael Haller, and Werner Hartmann. A generic framework for game development. In *In Proceedings of the ACM SIGGRAPH and Eurographics Campfire*, pages 3–8322, 2002.

[25] Michael Haller Fh, Michael Haller, and Werner Hartmann. A software architecture for games. In *University of the Pacific Department of Computer Science Research and Project Journal*, 2003.

[26] Eelke Folmer. Component based game development: a solution to escalating costs and expanding deadlines? In *Proceedings of the 10th international conference on Component-based software engineering*, CBSE'07, pages 66–73, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] Hector Garcia-molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* 2000.

[28] Julian Gold. *Object-Oriented Game Development.* Pearson Addison Wesley, 2004.

[29] Luke Hoban. F#: embracing functional programming in visual studio 2010. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 7:1–7:1, New York, NY, USA, 2010. ACM.

[30] Alan D. Hudson, Justin Couch, and Stephen N. Matsuba. The xj3d browser: community-based 3d software development. In *ACM SIGGRAPH 2002 conference abstracts and applications*, SIGGRAPH '02, pages 327–327, New York, NY, USA, 2002. ACM.

[31] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[32] Dan Liebgold. Adventures in data compilation. http://www.naughtydog.com/docs/Naughty-Dog-GDC08-Adventures-In-Data-Compilation.pdf, 2010.

[33] Herb Marselas. Profiling, data analysis, scalability, and magic numbers, part 2: Using scalable features and conquering the seven deadly performance sins. http://www.gamasutra.com/view/feature/3136/profiling_data_analysis_.php.

[34] Ian Millington. *Game Physics Engine Development (The Morgan Kaufmann Series in Interactive 3D Technology).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[35] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[36] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[37] Erick B. Passos, Jonhnny Weslley S. Sousa, Esteban Walter Gonzales Clua, Anselmo Montenegro, and Leonardo Murta. Smart composition of game objects using dependency injection. *Comput. Entertain.*, 7:53:1–53:15, January 2010.

[38] Jeff Plummer. A flexible and expandable architecture for computer games. Master's Thesis.

[39] Guido Van Rossum and Phillip Eby J. Pep 342 - coroutines via enhanced generators. http://www.python.org/dev/peps/pep-0342/, 2010.

[40] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS.* North, 1990.

[41] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[42] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 31–42, New York, NY, USA, 2007. ACM.

[43] Walker White, Christoph Koch, Johannes Gehrke, and Alan Demers. Better scripts, better games. *Queue*, 6:18–25, November 2008.

[0]