

Formal Specification

April 28, 2004
Version 0.10

Robbie Hott
Dept. of Comp. Sci.
The College of William and Mary
Williamsburg, VA 23185
jrhott@cs.wm.edu

Sahil Mithal
Dept. of Comp. Sci.
The College of William and Mary
Williamsburg, VA 23185
sxmith@cs.wm.edu

© Copyright April 2004
All Rights Reserved

Contents

1	Scope	2
1.1	Specifications	2
1.2	General Abstractions	2
2	Conventions and Notation	3
3	Type Extensions to Z	4
3.1	Real Number Arithmetic	4
3.2	Real-Based Types	5
4	Fundamental Billiard Types	6
4.1	Definitions	6
4.2	Coordinate	6
4.3	Ball	6
4.4	Pocket	6
4.5	Rail	6
4.6	PoolTable	6
4.7	Rack	7
4.8	ShotData	7
5	Complex Billiard Types	8
5.1	TableState	8
6	Operations	9
6.1	Operations	9
6.2	Rack the balls	9
6.3	Shooting the balls	10
6.4	Pocketing the balls	10

1 Scope

1.1 Specifications

This formal specification attempts to specify the operations of the pool table in a game of billiards. In this specification, we address three main operations on the table: racking the balls, taking a shot, and pocketing a ball; along with all of the functions and types that are required by them.

Racking the balls is defined in the method *rackBalls*. In this method, we are given a rack with the balls that should appear on the table in that rack. The user determines the order and positioning of the balls in the rack and passes it to the method. *rackBalls* checks the constraints to ensure that the rack is good, and then places the balls on the table in that racked formation.

Taking a shot is defined in the method *takeShot*. In the method, we are given the data for the shot, and we use the abstract function *calculateNewPosition* to determine the new positions of the balls on the table after applying the shot information given. *takeShot* then determines that the pool table still passes all of the constraints necessary for the game.

Pocketing a ball is defined in *pocketBalls*. This is a very simple method, which takes a set of balls and a set of pockets and if any balls significantly overlap onto a pocket, they fall in, and are mapped in the *pocketBall* function in the *TableState*.

1.2 General Abstractions

In this formal specification, we are abstracting many of the complex and user decided issues of the billiards game. We are abstracting all of the math involved in computing shots, including movements of the balls through time and the collisions of each ball. The end result is determined by the function *calculateNewPosition*. We are also abstracting the type of rack given to the system, how the balls are placed into the rack, and whether it is tight or loose because those are all up to the user of the system to determine. The size and shape of the pool table is also abstracted because it too is a user defined choice, depending on how they have their table built.

2 Conventions and Notation

This section defines the conventions and text styles used throughout this document. The notation and convention descriptions specific to the Z notation have been omitted.

defined term: A defined term is underlined.

variableName: Variable names begin with a lower case letter. Additional words in the variable name begin with capital letters and are concatenated.

TypeName: Type names begin with an upper case letter. Additional words in the type name begin with capital letters and are concatenated.

3 Type Extensions to Z

[Provided for your benefit. Delete it if you don't need it.]

3.1 Real Number Arithmetic

In this section we begin the formal specification with the definition of abstract system of real numbers and operations. Z does not provide a definition of real numbers, so we provide our own limited definition.

[\mathbb{R}]

$0_{\mathbb{R}} : \mathbb{R}$
 $1_{\mathbb{R}} : \mathbb{R}$

We introduce \mathbb{R} as a given type, and declare $0_{\mathbb{R}}$ and $1_{\mathbb{R}}$ to be elements of that type. In our use of real numbers, the subscript is used to distinguish between the values and operators used for non-reals and those used for reals. [Z doesn't have operator overloading.]

$- +_{\mathbb{R}} - : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 $- -_{\mathbb{R}} - : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 $- *_{\mathbb{R}} - : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 $- /_{\mathbb{R}} - : \mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{R}$
 $- =_{\mathbb{R}} - : \mathbb{R} \leftrightarrow \mathbb{R}$
 $- \neq_{\mathbb{R}} - : \mathbb{R} \leftrightarrow \mathbb{R}$
 $- <_{\mathbb{R}} - : \mathbb{R} \leftrightarrow \mathbb{R}$
 $- >_{\mathbb{R}} - : \mathbb{R} \leftrightarrow \mathbb{R}$
 $- \leq_{\mathbb{R}} - : \mathbb{R} \leftrightarrow \mathbb{R}$
 $- \geq_{\mathbb{R}} - : \mathbb{R} \leftrightarrow \mathbb{R}$
 $+ /_{\mathbb{R}} : \mathbb{F} \mathbb{R} \rightarrow \mathbb{R}$
 $- *_{f(\mathbb{R})} - : (\mathbb{R} \rightarrow \mathbb{R}) \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$
 $- +_{f(\mathbb{R})} - : (\mathbb{R} \rightarrow \mathbb{R}) \times (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$
 $- *_{pf(\mathbb{R})} - : (\mathbb{R} \leftrightarrow \mathbb{R}) \times \mathbb{R} \rightarrow (\mathbb{R} \leftrightarrow \mathbb{R})$
 $- +_{pf(\mathbb{R})} - : (\mathbb{R} \leftrightarrow \mathbb{R}) \times (\mathbb{R} \leftrightarrow \mathbb{R}) \rightarrow (\mathbb{R} \leftrightarrow \mathbb{R})$
 $intToReal : \mathbb{Z} \rightarrow \mathbb{R}$
 $intToReal 0 = 0_{\mathbb{R}}$
 $intToReal 1 = 1_{\mathbb{R}}$
 $dom(- /_{\mathbb{R}} -) = \mathbb{R} \times \mathbb{R} \setminus \{0_{\mathbb{R}}\}$
 $\forall x, y : \mathbb{Z} \bullet intToReal x = intToReal y \Leftrightarrow x = y$

We introduce type definitions for functions that operate on real numbers and functions of real numbers, abstracting the definitions. We first declare addition and division as an infix functions from pairs of reals to reals. Next we define the “distributed summation” operator, which computes the sum of a finite set of reals. We then define four operators for performing the distributed summation and product of both total and partial functions on reals. The function *intToReal* is used to map integers to reals, similar to a cast in programming languages. [Z doesn't have implicit casting.]

3.2 Real-Based Types

$$\textit{Probability} ::= \{p : \mathbb{R} \mid 0_{\mathbb{R}} \leq_{\mathbb{R}} p \leq_{\mathbb{R}} 1_{\mathbb{R}} \bullet p\}$$

We define a probability as a real number between the values of 0 and 1 inclusive.

$$\textit{Rate} ::= \{r : \mathbb{R} \mid r \geq_{\mathbb{R}} 0_{\mathbb{R}} \bullet r\}$$

We define a rate as a real number greater than or equal to 0.

$$\textit{Time} ::= \{t : \mathbb{R} \mid t >_{\mathbb{R}} 0_{\mathbb{R}} \bullet t\}$$

We define time to be a real number greater than 0.

4 Fundamental Billiard Types

4.1 Definitions

$BALLMASS : \mathbb{R}$
$BALLRADIUS : \mathbb{R}$
$POCKETRADIUS : \mathbb{R}$

$BALLMASS$ and $BALLRADIUS$ are given global constants that define the mass and radius of the balls on the table. $POCKETRADIUS$ is a given global constant that specifies the radius of a pocket.

4.2 Coordinate

[*Coordinate*]

Coordinate is a given type that represents points in space.

4.3 Ball

<i>Ball</i>
$mass : \mathbb{R}$
$radius : \mathbb{R}$
$position : \textit{Coordinate}$
$mass = BALLMASS$
$radius = BALLRADIUS$

Ball is a type that represents the billiard balls. We expand it here so that the mass and radius of the balls can be displayed, as well as including the *Coordinate*, which is the point at which the ball is located.

4.4 Pocket

<i>Pocket</i>
$radius : \mathbb{R}$
$position : \textit{Coordinate}$
$radius = POCKETRADIUS$

Pocket is a type that represents the pockets on the pool table. Similar to the *Ball* class, we expand it here to show that it contains a point at which it is located and that it has a radius.

4.5 Rail

[*Rail*]

Rail is a given type that represents rails.

4.6 PoolTable

PoolTable

pockets : \mathbb{F} *Pocket*
rails : \mathbb{F} *Rail*
area : \mathbb{F} *Coordinate*

#pockets = 6
#rails = 18

PoolTable contains information about the pool table and the layout. It includes a set of pockets which are the pockets on the table, a set of rails to go around the table, and an area, which is the set of all coordinates in the table space.

4.7 Rack

Rack

balls : \mathbb{F} *Ball*
area : \mathbb{F} *Coordinate*

$\forall ball : balls \bullet ball.position \in area$

Rack is a type that represents the rack of the balls. It defines the order and placement of the balls given in the set of *Ball*, as well as the area in which the balls reside, given by the set of coordinates. The set of coordinates which are in a rack must be a subset of the coordinates in the *PoolTable* because the rack must reside on the table.

4.8 ShotData

[*ShotData*]

ShotData is a given type that represents the data given to the system at the time a shot is taken. It includes items such as the velocity, angle, power, etc, which are needed to calculate a taken shot. This will be used in a function to determine the new placements of all the balls after a shot is taken, but we abstract the details of the actual shot data.

5 Complex Billiard Types

5.1 TableState

TableState

balls : \mathbb{F} *Ball*

table : *PoolTable*

pocketBall : *Ball* \leftrightarrow *Pocket*

$\text{dom } \textit{pocketBall} \subseteq \textit{balls}$

$\text{ran } \textit{pocketBall} \subseteq \textit{table.pockets}$

TableState is the current state of the table at a given time. It contains a finite number of balls, mainly 16 for 8-ball and 10 for 9-ball. It also contains the *PoolTable* which is the layout of the table and the pockets included in it. Lastly, there is a function that maps from *Ball* to *Pocket* in which a *Ball* element is added when it enters a pocket in the *table*.

6 Operations

6.1 Operations

In this section, we define some of the functions needed to calculate the information needed to perform actions on the pool table.

The *getDistanceTo* function gets the distance (as a Real Number) between two Coordinate points. We abstract the method of actually computing the distance between the two points, but it can be easily obtained using the Euclidian Distance formula for 2D distances.

$$\left| \begin{array}{l} \text{getDistanceTo} : \text{Coordinate} \times \text{Coordinate} \rightarrow \mathbb{R} \end{array} \right.$$

The *doesNotOverlap* relation is a relation between two balls. If a ball doesNotOverlap another ball, then it means that the distance between them is more than 2 of their radii apart, so they are touching or are farther apart, but not overlapping.

$$\left| \begin{array}{l} \text{doesNotOverlap} : \text{Ball} \leftrightarrow \text{Ball} \\ \hline \forall \text{ball1} : \text{dom doesNotOverlap} \bullet \forall \text{ball2} : \text{ran doesNotOverlap} \mid \text{ball1} \neq \text{ball2} \bullet \\ \text{getDistanceTo}(\text{ball1.position}, \text{ball2.position}) \geq_{\mathbb{R}} (\text{ball1.radius} +_{\mathbb{R}} \text{ball2.radius}) \end{array} \right.$$

The *significantlyOverlaps* relation is similar to the *doesNotOverlap* relation, except in the opposite direction. A *Ball* and a *Pocket* are in this relation if the center of the ball is at the radius of the *Pocket*, which means that the *Ball* is actually in the *Pocket*.

$$\left| \begin{array}{l} \text{significantlyOverlaps} : \text{Ball} \leftrightarrow \text{Pocket} \\ \hline \forall \text{ball} : \text{dom doesNotOverlap} \bullet \forall \text{pocket} : \text{ran doesNotOverlap} \bullet \\ \text{getDistanceTo}(\text{ball.position}, \text{pocket.position}) \leq_{\mathbb{R}} (\text{pocket.radius}) \end{array} \right.$$

calculateNewPosition is a function that maps a *PoolTable*, a *Ball* and a *ShotData* to a *Ball*. This function is used to abstract all of the math needed for shots to be taken, as well as all of the collisions between balls, a ball and a rail, and stops a ball if the collision is between a ball and a pocket. For each *Ball* on the table, we will use this function to calculate it's new position for after the shot has been taken.

$$\left| \begin{array}{l} \text{calculateNewPosition} : \text{PoolTable} \times \text{Ball} \times \text{ShotData} \rightarrow \text{Ball} \\ \hline \forall \text{ball1}, \text{ball2} : \text{ran calculateNewPosition} \mid \text{ball1} \neq \text{ball2} \bullet \\ \text{ball1} \text{ doesNotOverlap } \text{ball2} \end{array} \right.$$

6.2 Rack the balls

The function presented here describes the act of racking the balls. The user, or in the case of the system, the Rules Engine gives this function a *Rack* which has a set of balls (ordered and positioned) into their area, which is the rack. This can either be a tight rack, the balls are close, or a loose rack, the balls are not close and the area of the rack is large enough to allow for extra room between balls.

$rackBalls$ $\Delta TableState$ $rack? : Rack$
$rack?.area \subset table.area$ $\forall ball1, ball2 : rack?.balls \mid ball1 \neq ball2 \bullet ball1 \underline{doesNotOverlap} ball2$ $balls' = rack?.balls$ $table' = table$ $pocketBall' = pocketBall$

In this definition, the *TableState* is changed, and basically initialized, with the balls given to it in the *Rack*. It uses this information to place the balls on the table, provided that they do not overlap. The actual table does not change, and neither do the balls that are pocketed because they are all in the rack.

6.3 Shooting the balls

The *takeShot* method is the most complicated method of this specification. It takes as input a *ShotData* given by the user, which includes all the information needed to effectively apply the shot to the table. Then, we use the function defined earlier to do all the calculations of the new ball positions and place them back on the table. This effectively covers all of the physics involved, which are abstracted away.

$takeShot$ $\Delta TableState$ $shotData? : ShotData$
$\forall ball : balls \bullet balls' = balls' \cup \{calculateNewPosition(table, ball, shotData?)\}$ $table' = table$ $pocketBall' = pocketBall$ $\#balls' = \#balls$ $\forall ball : balls' \bullet \forall pocket : table.pockets \mid ball \underline{significantlyOverlaps} pocket \bullet$ $pocketBall' = pocketBall' \cup \{ball \mapsto pocket\}$

After the shot has been taken, we must require some constraints on the table, basically that the number of balls has not changed, since we cannot remove balls from the table, nor add new ones. Also, we must be sure that the balls do not overlap, but that is a requirement of the *calculateNewPosition* function itself, which is held through in *takeShot*. If a ball is close enough to a pocket, we add a mapping of it to the *pocketBall* function because the ball has been pocketed.

6.4 Pocketing the balls

To pocket a ball, the system takes the set of balls on the table and checks to see if they overlap significantly with a pocket. If they do, then they are added to the mapping, effectively setting the balls as pocketed.

pocketBalls

Δ *TableState*

$table' = table$

$balls' = balls$

$\forall ball : balls \bullet \forall pocket : table.pockets \mid ball \underline{significantlyOverlaps} pocket \bullet$
 $pocketBall' = pocketBall' \cup \{ball \mapsto pocket\}$