Troll, a Language for Specifying Dice-Rolls

Torben Ægidius Mogensen*
DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen O, Denmark
torbenm@diku.dk

ABSTRACT

Dice are used in many games, and often in fairly complex ways that make it difficult to unambiguously describe the dice-roll mechanism in plain language.

Many role-playing games, such as Dungeons & Dragons, use a formalised notation for some instances of dice-rolls. This notation, once explained, make dice-roll descriptions concise and unambiguous. Furthermore, the notation has been used in automated tools for pseudo-random dice-rolling (typically used when playing over the Internet).

This notation is, however, fairly limited in the types of dice-rolls it can describe, so most games still use natural language to describe rolls. Even Dungeons & Dragons use formal notation only for some of the dice-roll methods used in the game. Hence, a more complete notation is in this paper proposed, and a tool for pseudo-random rolls and (nearly) exact probability calculations is described.

The notation is called "Troll", combining the initial of the Danish word for dice ("terninger") with the English word "roll". It is a development of the language Roll described in an earlier paper. The present paper describes the most important features of Troll and its implementation.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; G.3 [**Probability and Statistics**]: Distribution functions

General Terms

Languages

Keywords

Dice, Probability, Domain-Specific Languages

1. INTRODUCTION

The first thing to ask is: Why would you want a formal notation for specifying dice-rolls?

*Work partially funded by the Danish Research Council for Nature and the Universe project 'Application of Principles of Programming Languages'.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

There are several answers to this:

- Formal notation can give a concise and unambiguous description which can be used to communicate ideas between people. This is, for example, the main motivation for mathematical notation. Games that use dice can (and sometimes do) use formal notation to describe dice-rolls.
- A formal notation is machine readable, which enables use of tools that analyse specifications of dice-rolls. Two types of tools are especially relevant for dice-rolls:

  Internet dice-roll servers for playing games that involve dice online or by email. There are many dice-roll servers, but they are each limited to a small number of different types of dice-roll. With a universal notation, a dice-roll server can perform rolls to any specification.

  Probability calculators As with any random element used in games, players and game designers are interested in knowing the probability of each possible outcome. A tool that takes a specification of a roll and calculates this is, hence, quite useful.

The concept of using formal notation for dice is not new: One of the first games to use a variety of dice shapes and rolling methods was Dungeons & Dragons from 1974 [3]. The rules introduced a formal notation for dice consisting of the following elements:

`d`n describes rolling a single dice with sides numbered 1–n. A normal six-sided die is, hence, denoted by `d6`.

m`d`n describes rolling a m dice with sides numbered 1–n and adding up their result. `3d6`, hence, denote rolling three six-sided dice and adding them up to get a result between 3 to 18.

In spite of introducing this notation, most of the rulebook used an alternative, less explanatory notation: An interval of values was shown, and it was up to the player to figure out which dice should be rolled to obtain this interval. For example, "3–18" would denote rolling and adding three six-sided dice, while "2–20" would denote rolling and adding two ten-sided dice.

In later editions, the m`d`n notation was used more and more, and in the most recent editions, the interval notation is completely eliminated. The m`d`n notation is also used in

other role-playing games and has been extended to include addition, so you can write things like `2d8+1` or `d6+d10`. Nevertheless, this extension is far from enough to describe dice-roll methods used in modern role-playing games. Here are some examples of methods from popular games:

- To determine values for personal characteristics such as strength and intelligence, Dungeons & Dragons suggest several methods, one of which is to roll four six-sided dice, discarding the lowest and adding the rest.
- "The World of Darkness" [1] lets a player roll a number of ten-sided dice equal to his ability score. If any of these show 10, additional dice equal to the number of 10s are rolled. If some of these also show 10, this is repeated until no more 10s are rolled. At this point, the number of values exceeding 7 are counted to give the final value of the roll which, consequently, can be any number from 0 upwards.
- "Ironclaw" [4] lets a player roll three dice of different sizes (number of sides) determined by ability scores and count how many exceed a threshold determined by the difficulty of the task.
- "Silhouette" [13] lets a player roll a number of ten-sided dice equal to his ability score and selects the highest of these as the result.

An universal method for dice-rolls needs to be able to describe all of the above, and more. Any Turing-complete programming language can do this, but the result is not necessarily very concise or readable, and it may be impossible to analyse descriptions for such things as probability distributions. Hence, we propose a notation that extends the m`d`n notation from Dungeons & Dragons while being readable to non-programmers after a short introduction.

This is not my first attempt at doing so: In 1996, I proposed a notation called "Roll" [7], which attempted the same thing. While it was moderately successful (it was used to calculate probabilities when modifying rules for the new edition of "The World of Darkness" [1]), experiences showed that the notation was not as readable to non-programmers as it could be and that it lacked features to concisely describe some of the more complex rolls. To address this, Roll was over time modified and extended until it had little resemblance to the original. Hence, a new name "Troll" was chosen for the revised and extended language.

One of the key features of Troll (and Roll) is the ability to automatically analyse descriptions for probability. Naive calculation will in many instances be far too time-consuming, so a number of optimisations are used. These are described in section 4.

2. THE BASICS OF TROLL

When you roll several dice at the same dice, you don't normally care about any specific order, so it is natural to consider the result of a dice-roll as a *multiset*, i.e., a collection where the order doesn't matter but where multiplicity of elements do. Multisets of integers are used throughout Troll as the only data structure. Set operations like membership, union, intersection and difference extend in the obvious way to multisets. For details, see [17].

We will call a multiset of integers "a value". A value containing exactly one number will in some contexts be treated as the number it contains. Some constructs require singleton values and run-time errors will be generated if these are applied to non-singleton values. Likewise, some operations require positive or non-negative numbers or non-empty multisets and will generate errors if applied to something other than that.

Like in the original Dungeons & Dragons notation, `d`n means rolling a die with n sides numbered from 1 to n. However, m`d`n is the value (multiset) of m integers obtained by rolling m n-sided dice, and you need an explicit `sum` operator to add up the numbers. Hence, what in Dungeons & Dragons is written as just `3d6` will in Troll be written as `sum 3d6`.

The reason for this is that, unlike in Dungeons & Dragons, we need to do many other things to the dice than just add them. For example, in Silhouette [13], we need to find the maximum of n ten-sided dice, which we in Troll can write as `max` n`d10`. There are also operators for finding the m largest numbers in a value and for finding the minimum or m smallest numbers. For example, the method mentioned earlier for determining personal characteristics in Dungeons & Dragons can be written as `sum largest 3 4d6`.

When the number of dice rolled depends on in-game values (such as ability level), it may be useful to use a variable to represent the number of dice. For example, `max n d10` will take the number of dice rolled from the variable `n`.[1] Variables can be bound to values when running Troll or locally inside definitions. We will return to this later.

Anywhere a number can occur in Troll, you can use an expression instead, so it is, for example, perfectly legal (though not very sensical) to write `d d10`, which would roll a dice the number of sides of which is determined by a `d10`. An expression like `d(2d10)` will, however, cause a run-time error, as the `d` operator requires singletons as arguments. The usual arithmetic operators on numbers, too, are defined only on singleton arguments.

Like `d` can take a numeric prefix to specify the number of rolls, you can specify multiple rolls of more complex expressions by using the `#` operator: `6#sum 3d6` specifies a multiset of six numbers each obtained by adding three dice[2]. You can use set-like notation like `{3,4,3}` to build multisets. Union of values is done using the infix operator `@`. There has never been any need for multiset intersection or difference when defining rolls, so operators for these are not included in Troll.

2.1 Comparisons, conditionals and bindings

Some dice-roll mechanism, such as that described above for Ironclaw [4], require the rolled numbers to be compared against a threshold and counting of those that do.

Comparison is in Troll done by filters, that compare the elements of a value with a single number and only returns the elements where the comparison holds. For example, `3<4d6` rolls four `d6` and retains only those numbers n such that $3 < n$. You can then count the number of remaining elements using the `count` operator, which simply returns the number of elements in a value. So, an instance of the Silhouette system where you roll one d8 and two d10 and where the threshold is 5 can be written as `count 5<={d8,d10,d10}`. There are filters for $=$, $<$, $>$, $\leq$, $\geq$ and $\neq$. Note the asym-

[1]The space between `n` and `d` is required to separate the lexical tokens.

[2]Allowing any expression to be prefixed by a number causes ambiguity, hence the need for an explicit operator.

metric nature of filters: The left-hand operand must be a singleton and is never returned, while the right-hand operand is a value, part of which may be returned.

Filters can also be used in conditionals. The expression `if` c `then` e_1 `else` e_2 evaluates the expression c. If this evaluates to a non-empty value, e_1 is evaluated and returned, otherwise e_2 is evaluated and returned. Note that the semantics of filters ensures that, for example, x`<`y, where x and y are singletons, is non-empty exactly when $x < y$.

In some cases you want to do two operations on the same die. You can't just repeat the expression that rolls the die, as you will get two independent rolls. Instead, you need to locally bind the value of a roll to a name and refer to that name repeatedly. The syntax for this is x `:=` e_1`;` e_2. While this may look like an assignment, it is a local binding corresponding to, for example, `let` x `=` e_1 `in` e_2 in Haskell. A local bind can allow us to define the dice-roll for Backgammon, where two identical dice are doubled to give four identical numbers:

```
x := d6; y := d6; if x=y then {x,x,y,y} else {x,y}
```

2.2 Repetition

Troll has two constructs for repeated die-rolling. The simplest is the `repeat` loop, which repeats a roll until a condition holds and then returns the value of the last roll (the one that fulfils the condition). For example,

```
repeat x:=d10 until x>1
```

keeps rolling a d10 until the result is greater than 1. It will, hence, return a value in the range 2...10. Note that the variable `x` is locally bound, not assigned to, so a loop like

```
x:=0; repeat x:=x+1 until x=10
```

does not terminate, as the two bound `x`'s are different. The above is equivalent to

```
x:=0; repeat y:=x+1 until y=10
```

which is, clearly, not terminating. The only thing that can change the value of the bound variable in different iterations is if the expression has a random element, such as a `d`n subexpression. Basically, the exact same roll is repeated until the condition holds (if ever).

For World of Darkness [1], we want to collect all the dice rolled until a certain condition is fulfilled, so we need a different kind of loop. The `accumulate` loop works like the repeat loop, but instead of returning just the value that fulfils the condition, it returns the union of all the values up to and including this point. With this, the World of Darkness dice roll can be expressed as

```
count 7< N#accumulate x:=d10 until x<10
```

Note that, like with `repeat`, the bound variable only changes as a result of different values of random elements in the expression; all iterations perform the same actions until these result in a value that fulfils the condition.

2.3 Other features

The above sections describe only the essential features of Troll. There are many other features including `foreach` loops, removal of duplicates in a multiset, random selection of n elements in a multiset and even recursive functions. To see a full description of the language, go to the Troll web page [6].

3. DIFFERENCES FROM ROLL

The basic idea of operations on multisets is unchanged from Roll to Troll, so the changes have mainly been addition of more operators, changes to syntax and new loop structures. Below is a summary of the changes and the reasons for them:

- The dice-operator `d` was in Roll purely a prefix operator, so to roll several identical dice, you needed the `#` operator, i.e., `3#d6` instead of `3d6`. Troll added the m`d`n form to get closer to the familiar Dungeons & Dragons notation.
- Troll added the set-like notation with curly braces, where Roll required building multisets by using the `@` (union) operator. In particular, this has made specification of the empty collection easier, as you can write {} instead of, for example, `0#0`.
- Local binding in Roll used a `let-in` syntax like in ML or Haskell, but users found the assignment-like syntax easier to read and write, especially when several bindings are nested.
- Filters (comparisons) were prefix operators in Roll, so you would write `< 3 x` instead of `3 > x`. The motivation for the prefix syntax in Roll was to emphasise the asymmetric nature of filters, but users found it confusing.
- Roll had a more powerful loop structure that allowed changes in variables between iterations, but it was too complex for most users. Hence, it was replaced by the simpler `repeat` and `accumulate` loops, and recursive functions were added to handle the more complicated (and much rarer) cases.
- More predefined operators were added. Some were just abbreviations of common cases, such as `min` abbreviating `least 1`, while others would be impossible to emulate without using recursive functions. An example of this is `pick`, which picks n elements from a multiset.

As an example, the World of Darkness roll described earlier would in Roll be written as

```
count N# >7 let x = d10 in repeat if =10 x then d10 else 0#0
```

which is, clearly, less readable.

Nearly all changes have been motivated through discussions with or requests from users of earlier versions. Sometimes to make descriptions easier to write and read and sometimes to make it possible to at all specify a certain dice-roll method.

4. IMPLEMENTATION

Troll is implemented as an interpreter written in Standard ML (specifically, Moscow ML). Two semantics are implemented:

- A random-roll semantics, where the interpreter makes random samples of the described dice-roll.
- A probability semantics, which calculates the probability distribution of the possible outcomes of the described dice-roll.

The random-roll semantics is implemented as a fairly straightforward interpreter using a pseudo-random-number generator seeded by system time, so this will not be detailed further. The implementation of the probability semantics is a bit more interesting, so we will elaborate on this.

If we for the moment ignore loops and recursive functions, a probability distribution for a dice roll is a finite map from outcomes to probabilities, such that the probabilities add up to one. Loops and recursion can make the number of possible outcomes infinite and allow the possibility of non-termination with non-zero probability, so a finite map is insufficient. We will, nevertheless, use finite maps and deal with the infinity issue later.

We will write a finite map as a set of pairs of values and probabilities. For example, the distribution for `sum 2d2` is:

$$\{(2, 0.25),\ (3, 0.5),\ (4, 0.25)\}$$

There are, basically, two ways in which we can calculate a finite probablity map for a dice-roll definition:

1. We can from each subexpression produce a finite map for its possible outcomes and combine these to find a finite map for the outcomes of the full expression.
2. We can use Prolog-style backtracking to obtain all global outcomes one at a time and count these at the top-level.

We can call the first method *enumeration in space* and the second *enumeration in time*. They have different advantages and disadvantages:

- Enumeration in space needs to enumerate all intermediate values at the same time, so if there are more intermediate values than final values, you can use very large amounts of memory. An example is `sum` n`d10`, where there are $\mathrm{O}(n^9)$ possible values of n`d10` but only $\mathrm{O}(n)$ possible values for the sum[3]. Hence, enumeration in time will use only $\mathrm{O}(n)$ space, while enumeration in space will use $\mathrm{O}(n^9)$ space.
- Enumeration in time needs only keep track of one value at any given time (except at the top-level count), so you don't need very much space.
- Because enumeration in time looks at intermediate values one at a time, it can not recognize that it has seen a value before and will, hence, often repeat calculations that it has already done. Enumeration in space can combine identical values in the finite map by adding up their probabilities and, hence, avoid doing further calculation twice. For example, while n`d10` has $\mathrm{O}(n^9)$ possible values, enumeration by backtracking has to look at 10^n combinations. To find the distribution for `sum` n`d10`, enumeration in time has to look at 10^n multisets of n numbers and add these up using 9 additions for each. Enumeration in space will combine values for n`d10` so it only needs to add up $\mathrm{O}(n^9)$ multisets.

Though space costs more than time on modern computers, reduction from exponential time to polynomial time is worth an polynomial increase in space.

[3]To be precise, n`d10` has $\binom{n+9}{9}$ possible outcomes.

Even so, enumerating $\mathrm{O}(n^9)$ multisets to find the distribution for `sum` n`d10` requires a lot of time and space. Fortunately, we can do better by the following observation:

Since $sum(A \cup B) = sum(A) + sum(B)$, we can find the distribution for `sum` n`d10` by first finding the distributions for `sum` $(n-1)$`d10` and `sum d10` and combine these into a distribution for `sum` n`d10`. If we apply this recursively, we never need to store a distribution with more than $\mathrm{O}(n)$ values, since there are only $\mathrm{O}(n)$ possible outcomes for `sum` m`d10` where $m < n$.

To exploit such algebraic properties, Troll uses a non-normalised representation for distributions described by the following recursive data-type definition:

$$D \equiv M! \ + \ D \cup D \ + \ D \mid_p D \ + \ 2{\times}D$$

where M is a multiset of numbers and $0 < p < 1$. $M!$ denotes the distribution with only one possible outcome, which is M, $d_1 \cup d_2$ combines the outcomes of two distributions by union, $d_1 \mid_p d_2$ chooses between the outcomes of two distributions with probability p of choosing from the first, and $2{\times}d$ is an abbreviation of $d \cup d$.

We can translate this representation into finite maps by the function F below:

$$\begin{aligned}
&F(M!) = \{(M,\ 1)\}\\
&F(d_1 \cup d_2)\\
&\quad = \{(M_1 \cup M_2,\ pq) \mid (M_1, p) \in F(d_1),\ (M_2, q) \in F(d_2)\}\\
&F(d_1 \mid_p d_2)\\
&\quad = \{(M, pq) \mid (M, q){\in}F(d_1)\} \cup \{(M, (1{-}p)q) \mid (M, q){\in}F(d_2)\}\\
&F(2{\times}d)\\
&\quad = \{(M_1 \cup M_2,\ pq) \mid (M_1, p) \in F(d),\ (M_2, q) \in F(d)\}
\end{aligned}$$

By operating on this representation, we can exploit two kinds of algebraic properties of functions on multisets:

- A function f is *linear* if $f(A \cup B) = f(A) \cup f(B)$.
- A function f is *homomorphic* if there exists an operator $\oplus$ such that $f(A \cup B) = f(A) \oplus f(B)$.

Examples of linear functions include filters like `7<`. We can lift a linear function f to distributions in the following way:

$$\begin{aligned}
f(M!) &= f(M)!\\
f(d_1 \cup d_2) &= f(M_1) \cup f(M_2)\\
f(d_1 \mid_p d_2) &= f(d_1) \mid_p f(d_2)\\
f(2{\times}d) &= 2{\times}f(d)
\end{aligned}$$

Examples of homomorphic functions include `sum` ($\oplus$ is $+$), `count` ($\oplus$ is $+$) and `min` ($\oplus$ is min). We can lift a homomorpic function f to distributions in the following way:

$$\begin{aligned}
f(M!) &= f(M)!\\
f(d_1 \cup d_2) &= f(M_1) \hat{\oplus} f(M_2)\\
f(d_1 \mid_p d_2) &= f(d_1) \mid_p f(d_2)\\
f(2{\times}d) &= \oplus^2 f(d)
\end{aligned}$$

$$\begin{aligned}
M!\hat{\oplus}N! &= (M \oplus N)!\\
(d_1 \mid_p d_2)\hat{\oplus}d_3 &= (d_1\hat{\oplus}d_3) \mid_p (d_2\hat{\oplus}d_3)\\
d_1\hat{\oplus}(d_2 \mid_p d_23 &= (d_1\hat{\oplus}d_2) \mid_p (d_1\hat{\oplus}d_3)
\end{aligned}$$

$$\begin{aligned}
\oplus^2 M! &= (M \oplus M)!\\
\oplus^2 (d_1 \mid_p d_2) &= (\oplus^2 d_1) \mid_{p^2} ((\oplus^2 d_2) \mid_{\frac{(1-p)^2}{(1-p^2)}} (d_1\hat{\oplus}d_2))
\end{aligned}$$

where $\oplus$ is the operator for the homomorphism f, $\hat{\oplus}$ is $\oplus$ lifted to union-free distributions and $\oplus^2 d$ is an optimized version of $d\hat{\oplus}d$.

4.1 Local bindings

If we locally bind a value to a variable, as in `x := d6; x+x`, the two occurrences of `x` after the semicolon must always refer to the same value. Hence, in the expression `x+x`, the distribution for `x` must have only one possible outcome. So the local binding must normalise the distribution for `x` to a finite map and then evaluate `x+x` for each possible value and combine the results to a new finite map.

By normalising, we lose all of the optimisations of using a non-normalised representation, so local binding is one of the most costly operations in Troll.

We represent normalised finite maps as a special case of the non-normalised representation where there are no union nodes and where the left operand to a choice node is always of the form $M!$, i.e.: $N \equiv M! + (M! \,|_p\, N)$. Furthermore, the nodes of the form $M!$ are in strictly ascending order (using a lexicographic ordering on multisets).

4.2 Loops

The first observation is that since all iterations evaluate the same expression and the last such evaluation provides the result, the outcomes of a loop of the form

`repeat x :=` e `until` c

is a subset of the outcomes of e.

The way we handle `repeat` loops in Troll is to first calculate the distribution d for e and then rewrite d into a form $d_1 \,|_p\, d_2$, where all outcomes of d_1 fulfil the condition c and none of those of d_2 do. It is now clear that the distribution for `repeat x :=` e `until` c is d_1: Repetition is done until we have a result in d_1, regardless of how unlikely this is. There is a possibility $p = 0$, i.e., that none of the outcomes of e fulfil c. It would be possible to include nontermination as a possible value in distributions, but since dice rolls are intended to terminate, we instead report an error whenever there is a positive chance of nontermination.

An accumulating loop of the form

`accumulate x :=` e `until` c

can have an infinite number of possible outcomes. If we, like above, rewrite the distribution for e into the form $d_1 \,|_p\, d_2$, we find that the distribution d' for the loop can be defined by the equation

$$d' = d_1 \,|_p\, (d_2 \cup d')$$

This will not have a finite solution unless $p = 1$ or $d_2 = \{\}!$.

Instead of trying to work with infinite maps, we have chosen to approximate by unfolding the above equation a finite (but user-definable) number of times and then replacing the remaining recursive reference to d' by d_1. If rerolls have probability less than 1, we can get arbitrarily good approximations by increasing the unroll depth.

We are now left with the problem of rewriting d to $d_1 \,|_p\, d_2$ given the condition c. We note that the loop introduces a local binding, so we start by normalising d. We then find $d_1 \,|_p\, d_2$ by the following function:

$$\begin{array}{lcl}
C(M!) & = & M! \,|_p\, M! \quad \text{where} \\
 & & \quad c_M = \text{ the distribution for } c \text{ when } \mathtt{x}{=}M \\
 & & \quad p = 1 - E(c_M) \\
C(d \,|_p\, d') & = & (d_t \,|_{p2}\, d'_t) \,|_{p1}\, (d_f \,|_{p3}\, d'_f) \quad \text{where} \\
 & & \quad d_t \,|_q\, d_f = C(d) \\
 & & \quad d'_t \,|_r\, d'_f = C(d') \\
 & & \quad p1 = pq + (1-p)r \\
 & & \quad p2 = pq/p1 \\
 & & \quad p3 = p(1{-}q)/(1{-}p1)
\end{array}$$

$$\begin{array}{lcl}
E(\{\}!) & = & 1 \\
E(M!) & = & 0 \quad \text{if } M \neq \{\} \\
E(d \,|_p\, d') & = & p \cdot E(d) + (1{-}p) \cdot E(d') \\
E(d \cup d') & = & E(d) \cdot E(d') \\
E(2{\times}d) & = & E(d)^2
\end{array}$$

The $M! \,|_p\, M!$ in the first line may seem a bit curious, as it is equivalent to $M!$, but since p is significant for later calculations, we write the distribution in this redundant way. If there are several possible values for `x`, i.e., if the normalised distribution contains a choice, we calculate the split for each possible value and combine the results.

$E(d)$ is the probability that an outcome of d is the empty multiset. Since conditions are considered true when they evaluate to non-empty multisets, the probability of a condition being true is 1 minus the probability of it being empty.

4.3 Other optimisations

In most places where distributions are built, some local simplifications at the top-level of the tree-structure are attempted. An incomplete list of these is shown below.

$$\begin{array}{lcll}
d \,|_p\, d & \to & d & \\
d_1 \,|_1\, d_2 & \to & d_1 & \\
d_1 \,|_0\, d_2 & \to & d_2 & \\
d_1 \,|_p\, (d_1 \,|_q\, d_2) & \to & d_1 \,|_{p'}\, d_2 & \text{where } p' = p{+}q{-}pq \\
 & & & \\
d \cup d & \to & 2{\times}d & \\
\{\}! \cup d & \to & d & \\
M! \cup N! & \to & (M \cup N)! &
\end{array}$$

These add a small overhead to construction of trees, but can sometimes reduce their size dramatically.

5. ASSESSMENT

So, how well does Troll work?

5.1 The language

While Troll is considerably easier for non-programmers and programmers alike to use than Roll, people with no programming experience at all usually find it hard to write definitions that involve conditionals or loops, but expressions like `sum largest 3 4d6` or `count 7<5d10` are not usually any problem. People with a minimum of programming experience (such as experience from writing formulae in spreadsheets) can usually write definitions using a single loop or conditional and more experienced users can use all of the language.

Usually, people can read and understand definitions that are more complex than they can write.

While a number of game designers around the world use Troll to calculate probabilities, the notation has not been

adopted in any game rules texts. Any one game will usually only use one or two different dice-roll methods, so it is easier for the writers to use a specialised notation or just plain English. Nor has any Internet dice-server used Troll to allow users to describe rolls that are not pre-programmed. So, overall, the success of Troll has almost exclusively been in probability calculation, where there (to my knowledge) are no other similar tools.

5.2 The implementation

Due to the optimisations enabled by the non-normalised representation, calculating the probability distribution of most simple definitions is very fast. For example, calculating the distribution for `sum 50d10` takes about 0.1 second on my fairly old machine[4] and the calculation for `count 7<100d10` takes less than 0.01 second.

But when rolls combine a large number of dice and operations that do not distribute well over the non-normalised representation, calculations can take very long and use enormous amounts of memory. This can even happen for dice-roll systems used in actual games. For example, the game "Legend of Five Rings" [18] uses a roll mechanism that can be described in Troll as

```
sum (largest M N#(sum accumulate x := d10 until x<10))
```

where `M` and `N` depend on the situation. With `M = 3`, `N = 5` and the maximum number of iterations for `accumulate` set to 5, Troll takes nearly 500 seconds to calculate the result, and it gets much worse if any of the numbers increase.

If exact calculation of probabilities takes too long, the random-roll semantics of Troll can be used to generate a large number of samples which can be used to calculate statistics that approximate the probabilities.

6. RELATED WORK

Apart from the notation originated in Dungeons & Dragons [3], most examples of notation for dice-roll are used only in games from a single publisher. Wikipedia [16] lists a few examples of those. Besides Roll [7] and Troll, the only other attempt at defining a universal dice-roll notation that I know of is in a Usenet post from 1992 [14]. There are many similarities between this and Troll: Numbers are kept separate until explicitly summed or otherwise operated on and there are clear equivalences to the Troll operations `sum`, `d`, `#`, `least` and `largest` (but not to the more complex Troll features). A partial implementation of the language was implemented in a dice-roll calculator [15].

There are many examples of extending traditional languages with probabilistic choice operators, e.g., [9, 2, 12], but in the main these do not allow calculation of probabilities, only sampling at runtime.

Other languages are designed for calculating probabilities [10, 5, 8, 11]. None of these are specialised for defining dice-rolls, but some similarities to Troll exist. For example, the stochastic lambda calculus [11], like Troll, can be instantiated to both a sampling and probability calculation. The authors note that calculating probabilities for product spaces (which are similar to unions) can take very long time and discuss translating expressions into *measure terms* that keep the parts of a product space separate as long as possible. This has some similarities to using a non-normalised form, but the measure terms are more complex (and potentially more powerful) than the representation used in Troll. The main probabilistic construct in the stochastic lambda calculus is `choose` $p\ e_1\ e_2$, which is equivalent to the $d_1 \mid_p d_2$ form in the unnormalised representation in Troll.

The predecessor of Troll, Roll is described in a paper [7] that includes formal semantics for both sampling and probability calculation. An implementation using an unnormalised representation similar to the one described in this paper is described, but not all the optimisations described above were applied to Roll.

[4] 3.2GHz Pentium 4, 1.5GB RAM

7. REFERENCES

[1] Bill Bridges, Rick Chillot, Ken Cliffe, and Mike Lee. *The World of Darkness – storytelling system rulebook*. White Wolf Publishing, 2004.

[2] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *POPL'99*, pages 189–202. ACM Press, 1999.

[3] E. Gary Gygax and Dave Arneson. *Dungeons & Dragons*. Tactical Studies Rules, Inc., 1974.

[4] Jason Holmgren. *IRONCLAW: Anthropomorphic Fantasy Role-Play*. Sanguine Productions Ltd., 1999.

[5] Claire Jones. *Probabilistic Non-Deteminism*. PhD thesis, Edinburgh University, 1990.

[6] Torben Mogensen. Troll homepage. `http://www.diku.dk/~torbenm/Troll/`, 2008.

[7] Torben Æ. Mogensen. Roll: A language for specifying die-rolls. In *PADL'03*, pages 145–159. ACM Press, 2003.

[8] Sungwoo Park. *A Programming Language for Probabilistic Computation*. PhD thesis, Carnegie Mellon University, 2005.

[9] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *ICAI'01*, pages 733–740. Morgan-Kaufmann Publishers, 2001.

[10] D. Pless and G. Luger. Toward general analysis og recursive probability models. In *UAI'01*, pages 429–436. Morgan-Kaufmann Publishers, 2001.

[11] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL'02*, pages 154–164. ACM Press, 2002.

[12] N. Saheb-Djahromi. Probabilistic lcf. In *MFCS'78, LNCS 64*, pages 442–451. Springer Verlag, 1978.

[13] Marc A. Vézina and Paul Lippincott. *Silhouette CORE Roleplaying Core Rules*. Dream Pod 9, 2003.

[14] Coyt D. Watters. Dice equation language. `http://groups.google.com/group/rec.games.programmer/msg/3b623636cb36beaa`, 1992.

[15] Coyt D. Watters. Gcalc. `ftp://ftp.funet.fi/pub/doc/games/roleplay/programs/pc/gcalc32x.zip`, 1997.

[16] Wikipedia. Dice notation. `http://en.wikipedia.org/wiki/Dice_notation`, 2008.

[17] Wikipedia. Multiset. `http://en.wikipedia.org/wiki/Multiset`, 2008.

[18] Rich Wulf, Shawn Carman, Seth Mason, Brian Yoon, and Fred Wan. *Legend of Five Rings, 3rd Edition*. Alderac Entertainment Group, 2005.