Design Rationale for the Standard ML bindings of the RenderMan Interface (RI::ML)

Ánoq of the Sun, Hardcore Processing *

May 19, 2023

1 Introduction

This document gives a design rationale for the Standard ML bindings of the RenderMan Interface (a.k.a. `RI::ML`). These Standard ML RenderMan bindings tries to conform to the *"RenderMan Interface Specification version 3.2"*. However, during the creation of these bindings, information from the *"RenderMan Companion"* and *"Advanced RenderMan: Creating CGI for motion pictures"* has also been used in an attempt to make things as correct as possible.

2 This document in other versions

This design rationale is a available in the following different file formats:

- `http://www.HardcoreProcessing.com/pro/riml/designrationale.html`
- `http://www.HardcoreProcessing.com/pro/riml/designrationale.pdf`
- `http://www.HardcoreProcessing.com/pro/riml/designrationale.ps`

3 Organization of modules in the ML RenderMan Interface

The RenderMan Interface in ML is divided into 4 different invariant structures which should not change across RenderMan implementations in ML and which are all instantiated from functors. Each RenderMan implementation in ML will also have to create it's own 5th structure to implement the interface. The 4 invariant structures are the following:

- *Rt* contains all RenderMan types.
- *RI* contains all constants in the RenderMan Interface which are prefixed with *RI_* in the C bindings.

*©2000-2001 Ánoq of the Sun (alias Johnny Andersen)

- *RIE* contains all constants which are prefixed with *RIE_* in the C bindings.
- *RIFCs* contains all constants in the RenderMan Interface which are prefixed with *Ri* but are really not part of the functions that one would call directly in the RenderMan Interface. This structure is not intended to be used directly though. It is intended that the functions in this structure are simply included into the structure implementing all other functions of the RenderMan Interface which are prefixed with *Ri*. This can be done in ML by writing *open RIFCs*.

The 5th structure that RenderMan implementations will have to implement is the *Ri* structure. This structure must conform to the signature called *RENDERMAN_INTERFACE* which also includes all the functions already implemented in the structure *RIFCs*. A structure called *RIB* which implements the *RENDERMAN_INTERFACE* signature is included with the RenderMan ML bindings. The structure *RIB* is basically a RIB file generator. It is actually a good idea to *not* call a structure implementing the *RENDERMAN_INTERFACE* signature *Ri*. There are 2 main reasons for this:

1. It allows multiple RenderMan implementations to coexsist in the same ML program.
2. It is always a good idea to encapsulate ML code into a functor and it is recommended that functors using the RenderMan Interace are parametrized with a structure called *Ri* conforming to the *RENDERMAN_INTERFACE* signature. See the file *RiExample.sml* for an example of this.

4 real vs. Rt.Real.real etc.

One of the design decisions of *RI::ML* which have been quite hard is how to deal with the precision of integers and reals in ML's safe typesystem. There are a few options:

- Use *Real32.real* and *Int32.int*. This will enforce that the precision of reals and integers is 32 bits as required by the RenderMan specification. It also means that all client applications will have the 32 bit precision hardcoded into them even if the RenderMan specification changes.
- Create nested modules *Rt.Real* and *Rt.Int* to be *Real32* and *Int32* respectively. If the modules are made opaque (as I believe it should be if this option is chosen), then this can enforce the precision of reals and integers as required by the RenderMan specification without hardcoding it into client applications.
- Just use *real* and *int* - this means that it is up to the compiler what the precision of reals and integers are. It is assumed to be the precision which is most convenient for the compiler on the given platform.

I believe that using *real* and *int* directly is The Right Thing (TM) for the following reasons:

- Using *Real32* or *Rt.Real* for reals (and similarly for integers) would most likely require an explicit typecast in ML at every call to a RenderMan function, since it is common to use real and int directly in ML applications. At runtime this typecast would problably be free in most cases, since *real* and *int* would problably have 32 bit precision already, so the problem here is really that it makes programming to the RenderMan Interface cumbersome.

- I really think that ML compilers in general should allow one to configure things like the precision of *int* and *real* - problably by using compiler flags. And if hardware acceleration (of any kind) is to be feasable in ML, it should most certainly be possible to configure these things in the ML compilers.

However, I'll agree that it still might be worthwhile to consider *Rt.Real* and *Rt.Int*, but I think using *Real32* and *Int32* is out of the question.

5 RtBoolean, RtVoid, RtString and RtPointer

I don't believe that it is worth encapsulating the RenderMan types *RtBoolean*, *RtVoid* and *RtString* in ML. The types *bool*, *unit* and *string* are part of Standard ML - with an emphasis on Standard. The *RtPointer* type is an ugly hack compromising the type system completely. It is mostly needed in C because of the lack of some of ML's language features - such as parametrized types (i.e. ML's *'a* types [1].) and sum types with constructors (i.e. ML's *datatype* language construct).

6 RtColor, RtMatrix and RtBasis

RtColor is called *Rt.Color* in ML and is defined as *RealArray.array*. If an ML compiler does not have a *RealArray* module, one can easily be defined by using *real Array.array* as done in the `CompatibilityLayer project` found on this homepage. The reason for using an array is for allowing indexing and updating individual components in a simple way as (fast) constant time operations. Using a *real list* has been considered and would surely be easier sometimes, but it does not seem natural when accessing individual color components. Also remember that colors in RenderMan can have an arbitrary number of color components (set by the *RiColorSamples* option), so just using a 3-tuple or a record with r, g and b fields is not good enough.

RtMatrix and *RtBasis* are always 4x4 matrices in RenderMan, so it has been defined as a 4-tuple of 4-tuples of reals. Using a *real array* would have been easier to index and modify. I'm not sure what would be most efficient though, but my quess is that arrays could be more efficient for hardware accelerated matrix transformations and that the current tuple types could be more efficient for software implementations of matrix transformations. FIXME: But maybe it should be an array...

[1] Pronounced: Alpha type. Alpha types in ML are parametrized types - similar to (but much nicer than) templates in C++.

7 RtToken

In the ML bindings the *Token* type has been made opaque. This means that to convert from a string to a token, one is required to call *Rt.MakeToken*. *Rt.TokenString* converts the other way. To test if 2 tokens are the same, use *Rt.EqualTokens*. This will just compare 2 integers rather than 2 strings. There is also a function called *Rt.TokenInt* which returns an associated integer from a token. This integer is what gives RenderMan implementors in ML the ability to quickly test the value of a token. This is also possible by comparing addresses of built-in tokens in C. The approach in C is an ugly hack though and it is not possible to do in ML beacause of ML's typesafety. The approach taken in ML with an opaque token type and a function for getting access to an associated integer actually seems more "correct". It even allows one to create an array based dispatch-table for tokens, so that a real time RenderMan renderer becomes feasable to implement. In ML, one would also be able to compare 2 tokens efficiently if the tokens had been implemented by making a reference to the string or by making a *unit ref*. However this will not give the ability to implement the array based dispatch-table mentioned before. Lastly it should be noted that the function *Rt.MakeToken* takes a little time to execute - so it might not be the best idea to call it every frame in a real time animation.

8 RiObjectBegin, RiLightSource and RiAreaLightSource

In the ML bindings the function *Ri.ObjectBegin* has got an extra parameter of type *Rt.ObjectHandle option*. This means that either no object handle is passed to the function (by passing the value *NONE*) or that an object handle *h* is passed (by passing the value *SOME(h)*). If a handle is supplied to the function, then the handle must have been returned earlier by a call to *Ri.ObjectBegin*. This is actually somewhere in between the RenderMan Specification for the C bindings and the RIB bindings. In the C bindings there is no handle supplied to these functions and in the RIB bindings a handle is required.

The semantics in the ML bindings are that if no handle is supplied (*NONE*) then the RenderMan implementation creates a new handle (and writes it to the RIB file in case of a RIB generator). If a handle is supplied (*SOME(h)*) then the implementation is expected to use the handle given by erasing the object which previously used that object handle. This is the same as is done for *Ri.ObjectBegin* in RIB files.

Ri.LightSource and *Ri.AreaLightSource* works similarly, except that they take a *Rt.LightHandle option* as argument and return a *Rt.LightHandle*. However the RenderMan specification does not mention that this is legal to replace a light with some given light handle. So, strictly speaking this is a violation of the RenderMan specification... but it seems like a good way of doing things. Also, since handle numbers must be in the range [0, 65535] it will also make it easier to assert that one doesn't run out of handles because they can be reused. I believe this will be important when one is to implement for instance a realtime RenderMan compliant 3D game engine where many frames are rendered, and new light sources could be introduced often.

9 Functions passed as arguments

The function types *RtFilterFunc*, *RtErrorHandler*, *RtProcSubdivFunc*, *RtProcFreeFunc* and *RtAchiceCallback* has been changed into being a pair of a function and a name. This means that the name of any function can be passed into a RIB file. It can also be used to give certain functions (like the builtin ones) heavily optimized implementations. Part of the reason why this name is needed is that ML does not allow function eqiuvalence testing.

In C one can compare functions because it is really pointers to functions. This can could be done in ML too by creating references to functions. However, just as in C, this approach will not work if a function is given a new reference. Also, using references to functions in ML without an associated name won't allow the use of new function names in RIB files, which might be supported by some RenderMan renderers.

10 Things which have been removed in the ML bindings compared to the C bindings

- *RI_NULL*: This does not make much sense try to encapsulate in ML. In the RenderMan Interface, *RI_NULL* is only used for terminating argument lists. In ML we've got the *list* type and the *nil* constructor which are part of Standard ML - again with an emphasis on Standard.
- *RI_TRUE* and *RI_FALSE*: Just like the type *bool*, the constructors *true* and *false* are standard in ML.

11 Tuples vs. curried function parameters

Reasons to use tuples instead of curried arguments:

- It forces client programmers to remember all parameters for each function. Had the arguments been curried and had one forgotten some parameters, the function call would simply not have any effect. And since the RenderMan API is mainly a sideeffecting API, and that the results of the function calls are usually ignored - the typechecker of the compiler would not even complain! This would furhter lead to the fact that an error of forgetting curried parameters would not be discovered before at runtime.
- The RenderMan API contains functions which are very atomic in nature, so you would usually not need currying in the first place.

However, functions with a parameter list at the end might be changed so that they just take a parameter list in ML to give these functions a more uniform interface - but this would still give the problems described above. It even means that mandatory arguments can be omitted when calling the functions!

12 lists vs. arrays vs. vectors

I have still not decided whether to use lists, vectors or arrays in ML for the RenderMan Interface. I only have a few statements which I believe to be correct, but I've not examined any of it thoroughly, so it might be wrong:

- I believe that arrays or vectors will always be faster to traverse than lists.
- In ML, arrays and vectors *might* be more expensive to produce than lists.
- Taking the length of a list in ML is a linear time operation, where as taking the length of an array or vector is a constant time operation. If the RenderMan Interface in ML is implemented in a clever way, it should however not be necessary for the renderer to take the length of any of the lists - but for calling the bindings to the RenderMan C interface it will at least be necessary to make a copy of the list into an array.
- If the compiler could do approriate representation analysis, then the compiler might be able to convert lists into arrays in an even more effective way than if it had been programmed directly as arrays...

13 Functions which differ in ML from the C functions

- *Ri.Procedural*: Uses an ML datatype (called *'a Blind*) which defines 2 constructors. One constructor (*BlStings*) is for passing a list of strings to the *Ri.Procedural* function for the builtin functions *ProcDelayedReadArchive*, *ProcRunProgram* and *ProcDynamicLoad*. This means that the strings can be passed on to a RIB file. The other constructor (*BlAnything*) uses an *'a* type. Passing the data as an *'a* type is much nicer than using the typeless *RtPointer* in C. It enforces type safety and does not give any unintended limitations.
- *Ri.TrimCurve*, *Ri.Polygon*, *Ri.GeneralPolygon*, *Ri.PointsPolygons*, *Ri.PointsGeneralPolygons*: In the C version of the RenderMan Interface, these functions have an integer as their first argument. They don't have this argument in RIB files and it's not needed in the ML bindings either, since we can always get the length of a list in ML or traverse a list until the end.
- *Ri.TransformPoints*: In the C version of the RenderMan Interface this function has an integer argument specifying the number of points in the list. It's not needed in the ML bindings since we can just traverse the list to it's end. Also, since ML is a functional language this function returns a new list of points rather than modifying an imperative array as is done in the C bindings.
- *RiLastError*: This is a global variable in C. In ML it is a function called *Ri.GetLastError* returning an integer - as it really should be :)