



## Tim Sweeney Archive - Postings

<http://www.team5150.com/~andrew/sweeney>

November 8, 2007

# Contents

<b>1</b>	<b>Prologue</b>	<b>7</b>
<b>2</b>	<b>Posts</b>	<b>8</b>
2.1	[comp.graphics.algorithms] Re: BSP Tree Balancing . . . . .	8
2.2	[comp.graphics.algorithms] Re: Bump Mapping/Texture Mapping . . . . .	9
2.3	[comp.graphics.algorithms] Re: HYPERBOLIC MAPPING, complete explanation . . . . .	10
2.4	[comp.graphics.algorithms] Re: Sphere collisions with BSP	10
2.5	[comp.graphics.algorithms] Re: Shadow Volumes, BSP, and HSR . . . . .	10
2.6	[comp.graphics.algorithms] Re: Shadow Volumes, BSP, and HSR . . . . .	11
2.7	[rec.games.programmer] Re: Fractal Cloud Generation . . .	12
2.8	[comp.graphics.algorithms] Re: Optimized custom math library for Pentium under Watcom C . . . . .	14
2.9	[rec.games.programmer] Re: Elegant Crash in DirectX . . .	15
2.10	[sci.math] Closed-form solutions of iterated functions? . . .	16
2.11	[comp.graphics.algorithms] Re: Direct 3D HAL???? . . . . .	16

---

2.12	[comp.graphics.api.opengl] future OpenGL features? . . . . .	17
2.13	[comp.graphics.algorithms] Direct Fourier or DCT texture mapping? . . . . .	19
2.14	[comp.graphics.algorithms] Re: Is there a faster way to find the highest bit set in a 32 bit integer? . . . . .	20
2.15	[gclist] Games and Realtime Garbage Collection . . . . .	21
2.16	[gclist] Unreal . . . . .	24
2.17	[gclist] Instantaneous GC . . . . .	28
2.18	[TYPES] State of the art in dependent typing? . . . . .	29
2.19	[fa.haskell] Tetration operator in functional programming .	32
2.20	[fa.haskell] Higher-order function application . . . . .	33
2.21	[fa.haskell] Re: Tetration operator in functional programming . . . . .	35
2.22	[TYPES] Correspondence of Linear Logic & Geometric Algebra . . . . .	36
2.23	[TYPES] Better vector math using dependent types . . . . .	37
2.24	[D] Persistence . . . . .	41
2.25	[D] Int to string . . . . .	41
2.26	[D] printf . . . . .	42
2.27	[D] *sigh* . . . . .	44
2.28	[D] Operator overloading . . . . .	45
2.29	[D] Macros . . . . .	47
2.30	[D] Getters and setters . . . . .	48
2.31	[D] Your feelings . . . . .	50
2.32	[comp.lang.c] Re: Parity of a number . . . . .	51

---

2.33 [comp.lang.c] Re: Parity of a number . . . . .	51
2.34 [sci.math] Constructing the integers from set theory without equivalence classes . . . . .	52
2.35 [sci.fractals] The nth iteration of $f(x)=x^2+c$ . . . . .	52
2.36 [sci.math] Re: Hestenes' Geometric Algebra . . . . .	54
2.37 [comp.lang.functional] Extending Church Numerals to integers, rationals . . . . .	55
2.38 [comp.lang.functional] Re: Extending Church Numerals to integers, rationals . . . . .	56
2.39 [microsoft.public.dotnet.languages.csharp] C# for game development? . . . . .	57
2.40 [microsoft.public.dotnet.languages.csharp] Concatenate two arrays? . . . . .	58
2.41 [comp.lang.functional] Re: Interpolating plus, multiplication etc . . . . .	58
2.42 [TYPES] Re: Type inference and related research... . . . . .	60
2.43 [sci.math] Extending GCD,MOD operations to rationals . . . . .	62
2.44 [TYPES] Classifying the cardinalities of types . . . . .	62
2.45 [TYPES] Re: Classifying the cardinalities of types . . . . .	65
2.46 [comp.lang.functional] Re: Y combinator . . . . .	67
2.47 [slashdot] We need 64-bit TODAY . . . . .	68
2.48 [comp.lang.functional] Re: Why isn't Haskell mainstream?—A newbie's view . . . . .	69
2.49 [comp.lang.functional] Re: curried function calling . . . . .	69
2.50 [slashdot] Already there . . . . .	70
2.51 [comp.lang.functional] Re: curried function calling . . . . .	72

---

2.52 [fa.haskell] Implementing RefMonads in Haskell without ST,IO . . . . .	73
2.53 [fa.haskell] Re: Implementing RefMonads in Haskell without ST,IO . . . . .	74
2.54 [fa.haskell] class MonadPlus m => MonadRing m . . . . .	75
2.55 [comp.lang.functional] Re: Is Strictness a Hack? . . . . .	77
2.56 [comp.lang.functional] Re: Is Strictness a Hack? . . . . .	78
2.57 [fa.haskell] Re: Typesafe MRef with a regular monad . . . . .	80
2.58 [fa.haskell] Re: Safe and sound STRef [Was Implementing RefMonads in Haskell without ST,IO] . . . . .	81
2.59 [comp.lang.functional] Re: Is Strictness a Hack? . . . . .	82
2.60 [slashdot] A few notes . . . . .	83
2.61 [comp.lang.functional] Re: local variables don't cause side effects . . . . .	84
2.62 [comp.lang.functional] Re: local variables don't cause side effects . . . . .	85
2.63 [comp.lang.functional] Re: local variables don't cause side effects . . . . .	87
2.64 [comp.lang.functional] Re: local variables don't cause side effects . . . . .	90
2.65 [slashdot] This is lame . . . . .	91
2.66 [comp.lang.functional] Terminology: sgn, abs, normal, magnitude . . . . .	91
2.67 [TYPES] Type theory vs floating-point arithmetic . . . . .	93
2.68 [sci.math] Computational algebraic integers . . . . .	94
2.69 [comp.lang.functional] Re: result of heterogeneous union . . . . .	95
2.70 [sci.math.symbolic] Re: Repeated exponentiation $n@m$ . . . . .	96

---

2.71 [comp.lang.functional] Re: result of heterogeneous union .	96
2.72 [comp.lang.functional] Re: Python from Wise Guy's Viewpoint . . . . .	98
2.73 [comp.lang.python] Re: Python from Wise Guy's Viewpoint	98
2.74 [comp.lang.functional] Re: Object Identity . . . . .	99
2.75 [sci.math] Re: Ordering a Power Set . . . . .	100
2.76 [fa.haskell] Re: lifting functions to tuples? . . . . .	101
2.77 [sci.math] Universal set theory and three-valued logic . . .	102
2.78 [sci.math] Re: Universal set theory and three-valued logic .	103
2.79 [sci.math] Re: Universal set theory and three-valued logic .	104
2.80 [comp.lang.functional] Re: Perfect list shuffling: an interesting algorithm . . . . .	106
2.81 [comp.lang.functional] Re: Computing alpha-equality in pure lambda calculus . . . . .	107
2.82 [TYPES] Functors and free theorems . . . . .	107
2.83 [comp.lang.functional] Re: My take on Monads . . . . .	108
2.84 [comp.lang.functional] Combining lazy and eager evaluation of terms . . . . .	109
2.85 [TYPES] Combining lazy and eager evaluation of terms . .	111
2.86 [comp.lang.functional] Re: Combining lazy and eager evaluation of terms . . . . .	114
2.87 [comp.lang.functional] Re: Combining lazy and eager evaluation of terms . . . . .	114
2.88 [TYPES] Combining lazy and eager evaluation of terms . .	115
2.89 [TYPES] Parametricity with subtyping and a Top type . . .	118
2.90 [TYPES] Subtyping, extensional equality, and contravariance	119

2.91 [TYPES] semantics of 'quote' . . . . . 121

# **Chapter 1**

## **Prologue**

Tim Sweeney's postings to various discussion lists and such. Dates are removed so the section titles are not as cluttered.

# Chapter 2

## Posts

### 2.1 [comp.graphics.algorithms] Re: BSP Tree Balancing

When picking a splitter polygon, rank all polygons based on a function like this:

```
Score =  
  (  
    (FLOAT)Balance *      (FLOAT)(OurAbs(Front-Back)) +  
    (FLOAT)(100-Balance) * (FLOAT)Splits  
  );
```

Where:

Balance = 0-100, tendency to balance tree rather than minimize cuts, 15-25 works well.

Front = polygons in front Back = polygons in back Splits = polygons split

For typical complex objects, the polygon count only increases by 20-30% after BSP'ing.

-Tim Sweeney, Epic MegaGames, Inc.

## 2.2 [comp.graphics.algorithms] Re: Bump Mapping/Texture Mapping

*I've looked in Watt and Watt, FvD, Graphics Gems I-III, etc. and I have the original paper written by Blinn. However, I have yet to see an implementation of the original algorithm proposed by Blinn. Frankly, every time I've seen Bump Mapping mentioned in any text, it's just glossed over. I understand it pretty much, I just would like to see an actual implementation. I guess my real question would be, how do you find the partial derivative of the bump function when your "bump function" is just a b/w bump map with varying heights. This is the way I would like to implement this as opposed to procedural bump mapping.*

An easy way is to store your bump map as a set of 2D normal vectors. Then, you can find the angle of incidence each lightsource against it with a dot product. A bump map texture could be something like

```
typedef struct
{
    float UNormal; // 0=facing straight out
    float VNormal;
} BumpMapTexture[USize][VSize];
```

Of course, for space reasons, it would be best to store bump components as bytes or words.

You'd determine the UNormal from a greyscale texture by taking the X differences, i.e.  $(\text{Texture}[U+1][V] - \text{Texture}[U][V]) / \text{Max}$ , and the VNormal from the Y differences.

-Tim

2.2. [COMP.GRAPHICS.ALGORITHMS] RE: BUMP MAPPING/TEXTURE MAPPING

## 2.3 [comp.graphics.algorithms] Re: HYPERBOLIC MAPPING, complete explanation

This is a very clever trick. I was working on a method similar to this a few months ago, but dropped it after a few hours of experimentation because of temporal problems. It produced great stills, but moving textures jiggled. I might have been missing something, though. Have you run your texture mapper at realtime speeds? Does it have this aliasing problem?

-Tim

## 2.4 [comp.graphics.algorithms] Re: Sphere collisions with BSP

*Brian - Since it is a sphere, just compute the distance of the center(oid) of the circle to each of the surfaces. It is a SINGLE check for each sphere (you can ignore all of it's faces since you are using a sphere approximation)*

Though this seems to be the correct solution at first glance, it's only an approximation. It overstates collision with convex edges. To get an exact solution, you need to either use spherical subdivision or handle convex volume edges and vertices. A better approach is probably "Don't try to do that". :-)

-Tim

## 2.5 [comp.graphics.algorithms] Re: Shadow Volumes, BSP, and HSR

*If you have a 3-D BSP tree, and you walk it front-to-back, would it not be possible to compute a "shadow volume" for each polygon and determine if the next polygon to be rendered is within*

*it? If it is entirely within the shadow volume, then it is NOT visible and does not have to be rendered. If it is only partially within the SV then it can be \*clipped\* to the previous poly's SV, and only have it's visible parts drawn. That would make for a "perfect" 3-D engine with \*no overdraw\*. Right?*

I coded this and the major problems are:

1. The shadow volume BSP's become very lopsided, so processing an additional polygon becomes more of an  $O(n)$  operation than an  $O(\log n)$  operation.
2. You'll get cracks between adjacent polys unless you incorporate logic to build a seamless SVBS mesh (a slow and operation).
3. Building the SVBSP requires taking the `sqrt()` of the difference of two numbers which is (A) slow, and (B) numerically unstable. Given the number of times you have to do this per frame, the speed and precision problems are killer.

-Tim

## 2.6 [comp.graphics.algorithms] Re: Shadow Volumes, BSP, and HSR

*3. Building the SVBSP requires taking the `sqrt()` of the difference of two numbers which is (A) slow, and (B) numerically unstable. Given the number of times you have to do this per frame, the speed and precision problems are killer.*

*Where exactly do you need to take the `sqrt`?*

To generate a BSP which I could clip world polygons by, I built a solid 3D BSP by forming a set of partitioning planes for each visible polygon. Each partitioning plane contained the viewpoint and the line comprising the polygon's side. You just do this for all polygons in the world, in front-to-back order, until the screen is completely solid. To build those

planes from a point and a line, I took a cross product, and normalized it. (In retrospect, I think this would have worked without the normalization - DOH!) :) The numerical instability came from sliver polygons, where most of the precision was lost during the cross product. This problem could have probably been overcome, as well as cracking, and optimization, but my view volume renderer was abysmally slow in complex areas compared to simpler methods, so I didn't spend much time before moving on to other methods.

Regarding convolving the world with a sphere in order to simplify collision, that's a great idea. You could also convolve the world with a cube and end up with all-polygon geometry.

-Tim

## 2.7 [rec.games.programmer] Re: Fractal Cloud Generation

*Has anybody got information on how to generate realistic, fractal clouds for computer graphics? I would be interested in any algorithm or literature on this subject.*

Here is some code I snipped from the game I'm programming.

```
//
// Make a random tiled 2D fractal texture of size US*VS.
// Dest is an array of US*VS floats whose values range from 0.0 to 1.0.
//
void MakeTiledFractal(FLOAT *Dest, int Size)
{
    void *MemTop = GMem.Get(0); // this is a simple memory pool

    // Make sure Size is a power of two:
    if( Size&(Size-1) ) appError("Size not power of two");

    // Make wraparound table
```

2.7. [REC.GAMES.PROGRAMMER] RE: FRACTAL CLOUD GENERATION

```
INT* WrapU = (INT*)GMem.GetFast((Size+1)*sizeof(INT)); // snag some memory
INT* WrapV = (INT*)GMem.GetFast((Size+1)*sizeof(INT));
for( int i=0; i<Size; i++)
{
    WrapU[i]=i;
    WrapV[i]=i*Size;
}
WrapU[Size] = WrapV[Size] = 0;

// Init random index
int iRand = 0;

// Init base
Dest[0] = 0.6 * 0.5;

FLOAT Range = 0.6 * 0.5;
int Speed = Size;

// Descend through mesh
for( Speed=Size/2; Speed; (Speed /= 2, Range *= 0.5) )
{
    FLOAT *Dest0 = &Dest[0];
    FLOAT *Dest1 = &Dest[Speed*Size];
    for( int v=Speed; v<Size; v+=Speed+Speed )
    {
        FLOAT *Dest2 = &Dest[WrapV[v+Speed]];
        for( int u=Speed; u<Size; u+=Speed+Speed )
        {
            FLOAT Base =
                (
                    + Dest0[ u-Speed ]
                    + Dest0[WrapU[u+Speed]]
                    + Dest2[ u-Speed ]
                    + Dest2[WrapU[u+Speed]]
                ) * 0.25;

            // GRandoms->Random(seed) is just a random number
            // from 0.0 to 1.0.
            Dest1[u-Speed] = Base + Range *
                (-1.0 + 2.0 * GRandoms->Random(iRand+0));
        }
    }
}
```

## 2.7. [REC.GAMES.PROGRAMMER] RE: FRACTAL CLOUD GENERATION

```
        Dest0[u      ] = Base + Range *
            (-1.0 + 2.0 * GRandoms->Random(iRand+1));
        Dest1[u      ] = Base + Range *
            (-1.0 + 2.0 * GRandoms->Random(iRand+2));
        iRand += 3;
    }
    Dest0 += (Speed + Speed) * Size;
    Dest1 += (Speed + Speed) * Size;
}
}
GMem.Release(MemTop);
}
```

## 2.8 [comp.graphics.algorithms] Re: Optimized custom math library for Pentium under Watcom C

*All routines are optimized specifically for Pentium (Pentium, Pentium Pro, Pentium with MMX and Pentium Pro with MMX). Optimizations include code that hurts performance on 486(487?) and 387 processors, however such code is kept to a minimum (compared to some FPU code I have seen).*

*Most functions in the library are very close to optimal. Short operations, such as a vector dot product, suffer a lot of stalls due to the fact that there generally aren't many parallel operations in such short operations. However longer operations like matrix multiplies (equivalent to 3 or more parallel dot products) usually only have a few stalls of 1 or 2 cycles.*

You should check out Visual C++'s optimizer. It generates stall-free code for vector and matrix operations floating point ops. I used VC++ to create a 3-component vector class and defined all of the operators - add, subtract, scalar multiply, dot product, matrix transform, etc. This makes 3D math really easy - "trade ease of implementation for speed", I thought.

To my utter surprise, I checked the assembly code output by VC++ and

### 2.8. [COMP.GRAPHICS.ALGORITHMS] RE: OPTIMIZED CUSTOM MATH LIBRARY FOR PENTIUM UNDER WATCOM C

its floating point was perfectly pipelined for the Pentium.

-Tim

## 2.9 [rec.games.programmer] Re: Elegant Crash in DirectX

*Hope I'm posting to the right place. Does anyone know a way to elegantly crash in Win95 with DirectX running? I find that when my app crashes (Release version) while having control of the screen, the user has to reboot to get his/her screen back. Is there a way to cast and catch an exception which leads to a code fragment for closing down directx?*

```
void main(void)
{
    // Set up DirectX here and go into fullscreen mode.
    try
    {
        // Call your game here.
    }
    catch(...)
    {
    }
    // Shut down DirectX here.
}
}
```

This will catch most problems. Crashes that are due to calling DirectX with invalid parameters, and some other hairy system level Win95 crashes, won't recover this way, but the above is the best that one can do in Win95.

-Tim

## 2.10 [sci.math] Closed-form solutions of iterated functions?

A few years ago, while experimenting with some fractal math, I wandered upon to the identity:

if  $f(x) = 2x^2 - 1$ , then the  $n$ th iteration of  $f(x)$ , i.e.  $f(f(f(\dots x))) = \cos(2^n \arccos(x))$ .

This is easy to see from the identity:  $\cos(2x) = 2 * \cos(x)^2 - 1$ ,  $\cos(4x) = 2 * \cos(2 * \cos(x)^2 - 1)^2 - 1$  etc.

In general, if  $F(x) =$  the inverse of  $f(x)$ , the  $n$ th iteration of  $f(g(F(x)))$  is equal to  $f$ ( the  $n$ th iteration of  $g(F(x))$ ), where  $f(x)=\cos(x)$ ,  $F(x)=\arccos(x)$ , and  $g(x)=2x$ .

Interestingly, the general formula works for non-integer values of  $n$ , conveniently defining the value of the iterated function over *\*all\** numbers.

At the time, I was working with the Mandelbrot set, trying to find a closed-form solution for  $f(x) = x^2 + a$ , for any value of  $a$ . Though I was utterly unable to generalize the above solution for  $2x^2-1$  (which can easily be transformed to  $x^2-1/2$ ), I have always been curious as to whether a closed-form solution does indeed exist. Any takers?

-Tim Sweeney

## 2.11 [comp.graphics.algorithms] Re: Direct 3D HAL????

*Hi Does anyone know how to bypass Direct3D and get straight to the HAL? This is because I want to support hardware acceleration without using Direct3D.*

I was looking into this, and I don't think it's possible. In immediate mode, you build "packets" which specify vertices, texture coordinates, and stuff.

It looks very much like the D3D hardware driver just takes these packets and does the minimum translation needed to send it to the card. In other words, the HAL is the lowest layer whose inputs are hardware-independent. So, short of programming directly to a specific piece of hardware, there's no lower level in Direct3D.

Fortunately there doesn't seem to be much overhead in the D3D translation layer.

-Tim

## 2.12 [comp.graphics.api.opengl] future OpenGL features?

*IMO it's not very realistic to cite one shader in a game and start suggesting extensions would have improved performance, particularly when you consider that it's one arbitrary (and very impressive) example of many possibilities. If you need commodity hardware acceleration for those features then using more complex shaders could be considered a disadvantage for some time to come.*

If we make the assumption that the most high-performance 3D apps will be geared towards rendering \*realistic 3d environments\*, then Quake is a very good example of what's needed in the short term. I think that "realistic 3d environments" is a safe and necessary assumption, since that's the most taxing widespread use of 3d hardware, especially at the consumer level. This means specializing the library a bit, but that's a time honored tradition (much as special case CPU performance was improved by caching and instruction pipelining - things which don't help all apps, but which greatly help the most common apps).

In a realistic 3D environment, the main polygon-level capabilities that are sorely needed now are:

\* Multipass texture rendering, for:

### 2.12. [COMP.GRAPHICS.API.OPENGL] FUTURE OPENGL FEATURES?

- Texture maps
- Detail textures/microtextures
- Bump maps
- Light/shadow maps

\* Vertex lighting options for:

- Gouraud shading
- Specular highlighting

High end hardware already does the vertex lighting, so the main area that needs improvement in OpenGL for the next few years is multipass texture rendering, and more advanced texture options (like bump mapping). This is a good direction for OpenGL, because it lets us get a huge (i.e. 5X-10X) improvement in apparent detail, just by adding multiple simultaneous texture mapping support to 3d hardware - something that is well defined and much easier to do in hardware than, for example, improving straight polygon rendering speed by 5X-10X.

I think you do need one texture matrix per simultaneous texture map you're rendering (whether the multi-pass rendering is done with multiple passes, or it's done simultaneously). The Quake case of light maps aligned 16:16 on texture maps is too special case, especially when one wants to extend it to dynamic lighting/shadow maps, where different coordinate systems are favorable.

In reality, this just means that each polygon vertex needs one (x,y,z) value and multiple texture (u,v,r,g,b) values (pardon the probably incorrect representation, I'm a newbie to hardware, I'm referring to two texture coordinates and an RGB lighting value). So hardware just needs the ability to interpolate "n" pairs of (u,v,g) coordinates, when n is small, i.e. 1-4.

The only major question is - what blending operations are needed between the multiple texture sources, color sources, and screen? For guidance, one can just look at the coolest raytraced, radiosity rendered, and other computer generated scenes around, and you don't need that many operations to capture 99% of the most common elements:

## 2.12. [COMP.GRAPHICS.API.OPENGL] FUTURE OPENGL FEATURES?

- Shadowing (multiply texture map texel by light map texel)
- Bump mapping (multiply computed bump-map light value by light map texel). Bump mapping could be achieved by generating a very low res (i.e. 16x16) specular highlight texture at each triangle vertex, and doing an (ugh) 5-way (i.e. bilinear -> trilinear -> 4-way -> 5-way) interpolation
- Fogging (multiplying for attenuation then adding fog)
- Alpha blending with interpolated or texture-lookup alpha for transparent and reflective surfaces

There are probably a few others, but the number of supported combinations can be kept small, by limiting the support to options which are very useful in rendering realistic environments.

-Tim

## 2.13 [comp.graphics.algorithms] Direct Fourier or DCT texture mapping?

I was thinking about the logistics of writing a texture mapper which maps textures from 2D Fourier or DCT space, directly onto the screen. This approach would seem to have many advantages, such as the ability to perform anisotropic texture filtering, by convolving the texture with an anisotropic filter kernel (a simple operation since convolution in the texture domain corresponds to multiplication in the Fourier domain).

It's trivial to write a texture mapper which renders an  $n \times n$  texture directly from Fourier space using  $n^2$  multiplies per pixel. With an FFT operation a faster execution time should be possible, i.e.  $n * \log(n)$  for a FFT. However, that operation works for arbitrary pixels, and it seems that a much faster operation might be possible, by exploiting the fact that texture mapping involves tracing a spatially linear (though not necessarily temporally linear, due to perspective correction) path through a texture.

### 2.13. [COMP.GRAPHICS.ALGORITHMS] DIRECT FOURIER OR DCT TEXTURE MAPPING?

Has anyone tried this? Is anyone aware of past work on this topic? Does this seem like a good idea, or can someone with more experience in Fourier transforms find fault with this approach?

-Tim Sweeney, Epic MegaGames, Inc.

## 2.14 [comp.graphics.algorithms] Re: Is there a faster way to find the highest bit set in a 32 bit integer?

(bunch of cool routines deleted)

Another interesting approach is to use the FPU, which automatically computes the index of the highest bit, using something like this:

```
Input DD 0
Temp DD 0
Output DB 0 ; log 2 of input

fild [Input] ; 4 cycles?
xor eax,eax ; free
fst [Output] ; 3 cycles
mov eax,[Output] ; 1 cycle
and eax, ??? ; 1 cycle, the value of ??? is left as an exercise to the
reader. :)
shr eax, ??? ; 1 cycle
add eax, ??? ; 1 cycle adjust for exponent bias.
; The idea of the above is to extract the exponent.
```

This would always be about 11 cycles, and I reckon it would be faster than the other methods posted, because their average case performance suffers from branch misprediction penalties which are pretty expensive especially on PentiumPro class processors.

-Tim Sweeney, Epic MegaGames Inc.

2.14. [COMP.GRAPHICS.ALGORITHMS] RE: IS THERE A FASTER WAY TO FIND THE HIGHEST BIT SET IN A 32 BIT INTEGER?

## 2.15 [gclist] Games and Realtime Garbage Collection

Hi,

I'm a game developer who is interested in garbage collection. The most recent game engine I worked on (Unreal, see <http://www.unreal.com/>) contains a built-in scripting language quite similar to Java, that is based on garbage collection for memory management. See <http://unreal.epicgames.com/unrealscript.htm> for an overview of my little language – it's pretty simplistic, but it does implement some neat object-oriented features such as language-level support for dynamic state (a.k.a. mode) switching and scoping of functions.

My current garbage collector is based on a simplistic and non-optimized "mark and sweep" routine that isn't performed in realtime. Once every few minutes (when the game's level changes), I just go off and perform a full, no-conservative garbage collection pass. It's a simple routine, because I do it at a known point in my code where I know there aren't any references to objects on the stack, and I have a single root object through which all active references can be traced. Object orientation is cool for games and 3D environments – my root object is sort of a "pointer to the 3d world". My script language maintains class "meta-data" listing all variables and their references, so it is a simple process to determine where object references reside in other objects.

For reference, I typically have 10,000 separate objects in memory. A typical object is a few hundred bytes, and references a few other objects. About 80% of my objects are known to be unchanging at runtime, for example game data like sounds, music, texture graphics, and animation data.

Anyway, for my next game, I want a much more general-purpose, real-time garbage collector. I've read a few books on garbage collection, and looked at links on the web, so I have a general idea of what's possible. But I was wondering if anybody might give me guidance on some topics:

1. Since > 80% of my objects are constant at runtime, I can be clever

and precompute information about their internal references and cycles (since they're known not to change after construction), and not have to ever check them in realtime for my garbage collector. Is this a generally known / researched technique, or did I invent something here? :-)

2. Since I'm defining my own language, I thought of using a new "visitor" keyword (that puts constraints on an object-reference variable), that means "this local variable (or function parameter) refers to an object, but it's not allowed to 'gain roots', so you can't assign it to a global variable, instance variable, or static variable". This is analogous to the meaning of "const Typ\* Ptr" in C++, "Ptr points to an object, but you can't change it". The idea behind this is that it enables the compiler to detect that I don't need to do any garbage collection work on a temporary object. For example:

```
// Transform a point by a matrix.
// Here we promise not to "root" p or m.
void transformPointByMatrix( visitor point p, visitor matrix m )
...

// Do some stuff.
point p = new point(1,2,3);
matrix m = new matrix(1,2,3,4,5,6,7,8,9); // create a new 3x3 matrix.
transformPointByMatrix(p,m);
return;
```

In this case, assume that the transformPointByMatrix function is defined in some other module, so I can't compile-time inline it. But with the "visitor" keyword, the compiler \*can\* realize that the objects "p" and "m" can't possibly gain any roots in the code above, so it can be explicitly destructed, without any GC overhead. Whereas, without the "visitor" keyword, both "p" and "m" become garbage and the collector eventually has to deal with them.

The other thing I like about this "visitor" limiter is that it helps programmers avoid making stupid memory management mistakes, keeping objects referenced that shouldn't be. For example, think about Windows

screen painting "device context" objects, temporary objects which are passed into a function for rendering purposes. These objects are only safe/usable for the duration of that function, after which they become invalidated. So I'd like the language to make it \*impossible\* for a programmer to sneakily "save off a copy of the object".

Also seems neat for "secure client" languages like Java, where you might want to pass in a high-level object to a function, but prevent it from being retained.

Is this a theoretically sound technique?

3. I'm a huge OOP proponent, but one aspect of OOP that I'm still uncomfortable with is the concept of "object lifetimes". In C++, object lifetimes were explicitly controlled by the app (for better or for worse). But there are lots of cases where that's very desirable, for example:

- A class encapsulating the TCP socket. The TCP socket might be closed at any point, after which the socket object is invalid. Without explicit object lifetime management, you have to write a lot of special case code to prevent these invalid objects from being used wrong, for example `if (!i_havent_been_closed) {do normal logic} else {just ignore the call}`.

- GUI windows that might be closed at any time.

- Objects in a game. When the game logic determines their life is over (i.e. if you kill a player), you remove it from the world and you \*don't\* want any references to them to remain!

Has anyone found a way to implement "explicitly managed object lifetimes" in OOP soundly?

For example, I was thinking about a specialized kind of object with the semantics "all references to this object must be weak; the object isn't garbage collected; it must be explicitly destroyed; when destroyed, all weak references to it are set to NULL, and an optional notification function for each reference is called". These would involve some extra overhead, such as maintaining backpointers at all times, but it seems like it might work.

4. References between objects in my game (and in 3D environments in

general) tend to be clustered: they form groups, where objects within a group contain tons of references to other objects in the group, but objects in a group contain a fairly small number of references to \*other\* groups. So I've been trying to devise a GC algorithm that takes this into account. Any suggestions?

5. Can anyone point me to techniques I can use in my language / compiler to make garbage collection simpler or faster? I'm lucky to have freedom over designing the language, compiler, and runtime, so anything is possible. And I need to manage thousands objects in realtime and 60 frames per second, so performance is important.

6. Since this is a game, realtime performance is more important than anything else. I can afford to waste a certain percentage of available memory for garbage objects, and I can afford the time to maintain reference counts and incrementally scan to collect garbage, but I \*can't\* afford to have any long pauses – and here in game land, spending more than 1 msec on garbage collection per 15 msec frame is the limit of acceptability. So, what is the best incremental "do a little bit of GC work at a time" approach? Note that I don't have to worry about "references to objects stored on the stack", since I can design my game to do garbage collection once per frame, in my update loop where I'm 100% sure there are no references to object on the stack.

Thanks for any help!

-Tim

## 2.16 [gclist] Unreal

*My script language maintains class "meta-data" listing all variables and their references, so it is a simple process to determine where object references reside in other objects.*

*To satisfy the curious (and to cut down on suggestions about things you already know about), it would be interesting to tell the list from which source you drew your meta-objects; for instance, CLOS (e.g. *Art of the Meta-Object Protocol*.) or Smalltalk*

*or one of its variants, like the late lamented Actor from White-water which provided Borland (nee Inprise) with ObjectWindows.*

Actually, the way UnrealScript handles "metaclass data" is very cool! When I saw the Java language spec, I thought their handle of metaclass data was a big hack, i.e. "Let's define a bunch of fixed-length data structures in the .class file to describe the contents of this object". So, Java is a beautiful object-oriented language, that is stored in a great contradiction of hard-coded "C" style data structures.

In UnrealScript, each unique class is described in a unique object of class "UClass". The UClass contains the name of the class, a packaging info, a list of all functions, all consts, all variables, etc.

Functions are described by UFunction objects.

Variables are described by UProperty objects. There are specific UProperty objects for various data types:

UIntProperty (a 32-bit integer variable) UFloatProperty (a 32-bit floating point value) UStringProperty (a variable length string – I treat these as variables, \*not\* references to objects like Java does). UReferenceProperty (a reference to an object).

In addition, UProperties can be parameterized. For example, a fixed length array of integers of is parameterized by a UFixedArrayProperty object that contains a UIntProperty object. I plan to extend this to constrained generic types (like C++ templates) for our next project.

This ends up naturally exposing complete introspection to the language, so something like the Java reflection API isn't needed. Just access your "class" object and its "variable" objects directly...I wish Java worked that way.

But from the garbage collector's point of view, I just do a recursive mark-sweep of active objects, starting at the root. It's like (oversimplified a bit):

```
MarkSweep( UObject* Obj )
```

```
{
    if( Obj==NULL || Obj->AlreadyMarked )
        return;
    Obj->AlreadyMarked = 1;
    UClass* Cls = Obj->GetClass(); // Gets pointer to class metadata.
    for( UProperty* Prop=Cls->FirstProperty; Prop!=NULL; Prop=Prop->Next )
        if( Prop->IsAReferenceToObject )
            MarkSweep( (UObject*)((BYTE*)this + Prop->Offset) );
}
```

*Unreal definitely requires large (by general consumer standards) amounts of physical memory, and one of the questions you no doubt spend a lot of time considering is how much effort to go to to deal with the fact that the Win32 platforms do such a bad job of managing VM.*

;)

*I haven't done UnrealScript programming, so I'm not entirely certain quite what "compile time" means for you (as opposed to "link time" or "load time"), but... your compiler can fairly easily do "escape analysis" on code to determine what parameters passed to methods cannot result in capturing a reference, since you don't permit long-lasting closures or continuation capture. In the long run, your compiler will be better at it than most humans (and it will work with existing code).*

UnrealScript is like Java in this regard: it's based on the "open world" hypothesis: base interfaces are assumed to never change, but implementations can change and new classes can be added anywhere in the hierarchy without losing binary compatibility.

So there isn't any way to tell that "the object reference I'm passing to a function won't become rooted" except by actually calling it and seeing (because you might be calling a new subclass's version of the function that does something evil).

*Are you familiar with Eiffel? While we often think about such things (handles which need immediate reclamation) in storage management terms, that's not really always appropriate. Eiffel's precondition/postcondition model puts the issue of whether e.g. a window is open or closed into a much wider realm of correctness guarantee based on preconditions and postconditions. Now, Eiffel uses a lot of static analysis to make the condition checking useful, and Eiffel doesn't have a C-like separate compilation model either. But Meyer's contract-driven model of design is a powerful one. I doubt whether your target audience would enjoy such things, but if the bulk of what they are doing is re-using your basic UnrealScript objects (which is what I would suspect) then it may be practical.*

I've been reading up on Eiffel a lot recently and I'm actually really excited about the "programming by contract" possibilities (please don't tell any other game programmers, they'll laugh at me!)

Can you help me understand something about pre/postconditions? I understand the Eiffel syntax, but I don't understand how they relate to the compiler: are pre/postconditions actually analyzed and/or proven by the compiler? My C/C++ background leaves me pretty skeptical, guessing they'll just be translated into runtime assertions. With Unreal's distribution model (10 partner companies using the engine for creating their own games, and hundreds of enthusiasts programming modifications and releasing them on the internet [check out <http://www.planetunreal.com/>, it's cool what users are doing nowadays]), when it comes to issues like object lifetimes and referential integrity constraints, I try to avoid making any assumptions that the language can't catch at compile-time and give you an error message about...because people are doing too much crazy stuff with the code.

Thanks a lot for all the help, it's greatly appreciated!

-Tim

## 2.17 [gclist] Instantaneous GC

Hello,

Thanks for all the feedback everyone has given regarding optimizing a game programming language for garbage-collection efficiency. It has been a great help.

I've started prototyping my collector and its data structures, and will probably be seeking more advice once I get into all the hairy details. I'm having a hard time finding the right set of tradeoffs to make, regarding immediate vs incremental collection, and robustness of support for multi-threading.

Once idea I've always liked, despite its huge implementation hurdles, is "immediate garbage collection", where all references to objects are subject to a write-barrier and some logic that causes all objects to be collected the very instant they become garbage. I'm not talking about a simple reference counting scheme (which has pitfalls with cyclic references), I mean a fully general GC scheme that notices immediately when groups of objects in the rooted graph become disjoint.

I like the concept because I'm a big fan of deterministic behaviour, and I've always been a little uneasy about the idea of garbage objects "eventually" being cleaned up.

Ok, so here's what I've come up with:

1. Absolutely naive implementation: Each time a reference is updated or goes out of scope, start at the root object and do a full mark-and-sweep through all objects and determine which ones are garbage. Advantage: always instantaneous. Disadvantage: Incredibly inefficient.
2. Faster implementation: Assign each object a positive number, starting with 0 at the root. For each object, maintain for lists of references: (incoming, outgoing) x (to lower numbered objects, to higher numbered objects). Enforce a rule that every object always has at least one incoming reference from a lowered number object. Whenever a reference changes that breaks this rule, move the offending object to the end of the list (make it the highest numbered object), and readjust the other objects—

which may cause them to recursively fall to the bottom of the list. In this process, some objects that land at the bottom of the list will have no upward references, and those objects have immediately become garbage. The overhead of this is (on a 32 bit machine) 8 bytes per object + 24 bytes per reference variable. This has a lot of  $O(n^2)$  performance cases, for example many double-linked list implementations, but I think the algorithm can be restructured to never be worse than  $O(n)$ . Still this is impractically slow.

So, my question is: is there any reasonably efficient way to do instantaneous garbage collection, using a write-barrier function that can immediately determine that objects have become garbage?

-Tim

## 2.18 [TYPES] State of the art in dependent typing?

I'm a game programmer, but lately I found out about Haskell, Cayenne, and dependent type systems. As a beginner, I have a few questions about dependent types and proofs-as-types systems. This is for fun and enlightenment rather than important research, so please don't feel obliged to respond unless you enjoy doing so.

1. In propositional logic, one can use  $\{x:\text{Nat} \mid x < 10\}$  to express the subset of natural numbers less than 10. Hindley-Milner extensions have been proposed to support subset types, such as Sokolowski's:

<http://www.ipipan.gda.pl/~stefan/Papers/research.html>

This seems very useful; is there a good reason why this hasn't been adopted by languages like Haskell and ML?

2. In languages like Augustsson's Cayenne:

<http://www.cs.chalmers.se/~augustss/cayenne/>

One can model both programs and proofs in a uniform language. Why

isn't this an area of huge interest, i.e. with would-be Sun Microsystems trying to popularize new mainstream languages based on this technology?

While Java is receiving tremendous hype, it's mostly an incremental improvement over C++, whereas this stuff is \*revolutionary\*, having the potential to make commercial software far more reliable. It seems surprising there isn't widespread interest.

Is it that people are turned off by the theoretical undecidability of powerful dependent type systems? This doesn't seem too likely to cause real-world problems – reasonable programs tend to be easy for a typechecker to analyze.

3. Is there anything preventing the programs-with-proofs concept from being taken to the extreme? For example, building abstract definitions of:

- sets
- monads
- groups
- lambda calculi
- categories
- sorting functions

and concrete instances containing proofs that they obey the appropriate laws. I've seen a few assorted systems along these lines, but nothing universal. Has anyone attempted to build a comprehensive set of types and proofs for mathematical objects or categories this way?

4. I noticed the following analogy between ordinary types and dependent types:

$$A \Rightarrow B \leftrightarrow \text{All}(x:A).B$$

```
A*B <-> Exists(x:A).B
A+B <-> -----
```

Can we fill in the blank meaningfully, i.e. is there a useful notion of "dependent sum"?

Would it be valid to say that "A+B" is not actually a primitive type itself, but instead corresponds to the existential subset type:

```
Exists( {t|t==A or t==B} ).t
```

Where the  $\{x|P(x)\}$  represents the notion of "subset types" as described above? One can read this as: there exists a type  $t$ , equal to either  $A$  or  $B$ , such that we encode a single value of type  $t$ . This encoding of sums is (I think) a more flexible version of Cardelli's "Any/TypeOf/ValueOf" definitions:

<http://research.microsoft.com/Users/luca/Papers/TypeType.ps>

5. Cayenne has a construct for defining records:

```
s = struct {a=1; s="hello";}
```

Has anyone tried defining such records as functions from strings (corresponding to tags) to dependent types? i.e.:

```
s :: (x::String).
  if      x="A" then Int;
  else if x="s" then String; -- else |_|

s = (x:String)->
  if      x="A" then 1;
  else if x="s" then "hello";
```

Such functions seem to obey the ordinary rules of records.

A variation on this theme is to represent certain records as "functions from types to dependent types". These seem to be signature-free records, which may be useful for composing Cayenne-style proofs without committing to specific proof names.

6. Looking at the bigger picture, are "programs with proofs" of practical value in future commercial software? I have to wonder if we're simply trading "bugs in programs" for "bugs in proof types".

For example, I can easily write a "qsort" function, test it with sample data, and be pretty confident it's ok.

But, what is the "proof type which all sorting functions must inhabit"? I have no idea, but suspect it's very complex and dependent on other proofs such as those governing ordering relationships. Given human fallability, it seems just as likely to write the wrong "proof type" (hence prove the wrong thing) as write an buggy program. Is this a problem in practice?

Thanks,

-Tim

## 2.19 [fa.haskell] Tetration operator in functional programming

There is a neat isomorphism between mathematics and type theory. Using  $/ =$  to indicate isomorphism, we have (roughly):

sum of numbers  $/ =$  disjoint union of types  
product numbers  $/ =$  cartesian product of types  
exponentiation  $/ =$  function spaces  
"sigma" summations  $/ =$  dependent products  
"pi" products  $/ =$  dependent functions  
successor function  $/ =$  "maybe" monad

In mathematics, there is a progression between the operators of addition, multiplication, exponentiation, etc., known as the Ackermann hierarchy.

The next operation in the sequence is often called "tetration" and refers to iterated exponentiation. Let us write  $a^b$  for "a raised to the power of b", and then  $a \rightarrow b$  for b raised to its own power a times, i.e.:

$$1 \rightarrow b = b \quad 2 \rightarrow b = b^b \quad 3 \rightarrow b = (b^b)^b \dots$$

Now I am trying to understand the type-theory analogy to tetration. Given "unit" as the one-valued type, "bool" as the two-valued type, "three" as the three-valued type (isomorphic to "maybe bool"), etc., clearly:

$$\text{unit} \rightarrow t / = t \quad \text{bool} \rightarrow t / = t \rightarrow t \quad \text{three} \rightarrow t / = (t \rightarrow t) \rightarrow t \quad \text{four} \rightarrow t / = ((t \rightarrow t) \rightarrow t) \rightarrow t$$

So, what is going on here? "bool  $\rightarrow$  t" is the type of identity functions on t; "three  $\rightarrow$  t" is especially interesting because it is the type of fixpoint operator from t  $\rightarrow$  t to t.

This hints that there may be some computational utility here. Is there a useful type-theory interpretation of tetration?

And is there a useful interpretation of tetration on infinitary types, like lists, i.e.  $\text{list}(\text{nat}) \rightarrow \text{nat}$ ? In particular, I am interested in defining the corresponding value-level operation whose type corresponds to a tetrant. In other words, pardoning the peculiar notation, fill in the blank:

- \* An injection  $\text{inl}(123)$  or  $\text{inr}("a")$  is of sum type  $\text{nat} + \text{string}$ .
- \* A pair  $(123, "a")$  is of product type  $\text{nat} * \text{string}$ .
- \* A lambda  $\lambda t \rightarrow t + 123$  is of function type  $\text{nat} \rightarrow \text{nat}$ .
- \* A \_\_\_\_\_ is of tetrant type  $t \rightarrow u$ .

-Tim

## 2.20 [fa.haskell] Higher-order function application

In Haskell, only a single notion of "function application" exists, where a function  $f :: t \rightarrow u$  is passed a parameter of type t, returning a result of type

u. Example function calls are:

```
1+2 sin 3.14 map sin [1:2:3]
```

However, a higher-order notion of function application seems sensible in many cases. For example, consider the following expressions, which Haskell rejects, despite an "obvious" programmer intent:

```
cos+sin – intent: \x-> ((cos x)+(sin x)) cos(sin) – intent: \x-> cos(sin(x))
(cos,sin)(1,2) – intent: cos(1),sin(2) (+)(1,2) – intent: (+)(1)(2) cos [1:2:3] –
intent: map cos [1:2:3]
```

From this intuition, let's postulate that it's possible for a compiler to automatically accept such expressions by translating them to the more verbose "intent" listed above, using rules such as:

1. Operator calls like (+) over functions translate to lambda abstractions as in the "cos+sin" example.
2. A pair of functions  $f::t \rightarrow u$ ,  $g::v \rightarrow w$  acts as a single function from pairs to pairs,  $(f,g)::(t,u) \rightarrow (v,w)$ .
3. Translating function calling into function composition, like "cos(sin)".
4. Automatic currying when a pair is passed to a curried function.
5. Automatic uncurrying when a function expecting a parameter of type  $(t,u)$  is passed a single value of type  $t$ .
6. Applying a function  $f::t \rightarrow u$  to a list  $x::[t]$  translates to "map f x".

I wonder, are these rules self-consistent? Are they unambiguous in all cases? Are there other rules we can safely add?

It also seems that every statement above is simply a new axiom at the type checker's disposal. For example, to describe the general notion of "cos+sin" meaning " $\lambda x \rightarrow (\cos(x)+\sin(x))$ ", we say:

for all types  $t,u,v$ , for all functions  $f,g :: t \rightarrow u$ , for all functions  $h :: u \rightarrow v$ ,  $h (f,g) = \lambda x \rightarrow h(f(x),g(x))$ .

Is this "higher order function application" a useful notion, and does any research exist on the topic?

-Tim

## 2.21 [fa.haskell] Re: Tetration operator in functional programming

*It should reflect arithmetic laws of tetration, which for example relate  $(b^{(a1*a2)})$  with  $b^{a1}$  and  $b^{a2}$  or so, but are there any? So how should  $(a,b) \rightarrow t$  be defined?*

I couldn't find any papers on this topic, so I'm trying to work out the laws now. The interesting ones are  $(a*b)^c$  and  $(a^b)^n$ . They don't seem to be as "obvious" as the rules for exponents and products, though.

There is also a notion of "dependent tetration" which is even more mysterious. Just as  $\text{Sigma}(a:t).b$  corresponds to "dependent sum" (with types or numbers) and  $\text{Pi}(a:t).b$  corresponds to "dependent product", there is a  $\text{Tetra}(a:t).b$  corresponding to "dependent tetration".

This operation is mysterious because exponentiation, the operator upon which tetration is defined, is the first operator in the Ackermann hierarchy which is not symmetric. So we can't just carelessly say that  $\text{Tetra}(a:t).f(a) = ((\dots \rightarrow f(e2)) \rightarrow f(e2))$  where  $e1, e2, \dots$  are the elements of type  $t$ . The result of  $\text{Tetra}(a:t).f(a)$  depends on the order in which we choose the elements of  $t$ . Which begs the question: \*which\* order is appropriate?

Perhaps our notion of dependent products and dependent functions is currently oversimplified, since we are neglecting this hidden "ordering" which is implicitly involved, but can neatly be hidden because sums and products are symmetric, up to isomorphism, i.e.  $a+b = b+a$  and  $a*b = b*a$ . I wonder, are there useful types (in a type theory) for which sum and product are not so neatly symmetric? One such construction is to apply the rules of simple matrix algebra, but put types inside the matrices rather than numbers. The resulting types commute:  $(a*b)*c = a*(b*c)$  but we don't have  $a*b = b*a$ , since the "shape" of the resulting matrix depends on the ordered dimensions of the inputs.

-Tim

## 2.22 [TYPES] Correspondence of Linear Logic & Geometric Algebra

While prototyping a compiler using Girard's Linear Logic as a type system and implementing, in it, a math library supporting the Geometric (Clifford) Algebra, I noticed a striking resemblance. Are these two systems known to coincide?

My conjecture: Linear Logic and Geometric Algebra are, in an axiomatic and semantic view, equivalent; as:

- The linear logic's "times" operator corresponds to the Geometric Algebra outer product.
- Linear "with" corresponds to the geometric product, with semantics analogous to the set-theoretic "exclusive or".
- "Par" matches the meet operator (the dual of the "times" and therefore of the outer product).
- "Plus" is the dual of the geometric product.
- "Nil",  $A \cdot 1$  corresponds to multiplication by the unit pseudoscalar.
- The "!" operator in linear logic appear to roughly correspond to the reverse of a multivector, and "?" to the conjugate of a multivector.
- The grade of a homogeneous multivector is analogous to the number of linear terms that appear in each conjunctive subterm of an expression. GA operations which raise, lower, and retain the grade, are analogous to linear logic with respect to the number of linear assumptions.
- The GA interpretation of linear implication "A -o B" is (surprisingly) not an exponential, but in fact the inner product, seeing that it is the algebras' sole grade- (uniqueness count-) lowering operation.
- The entire duality of linear logic (between conjunction and disjunction, and so on) is carried out identically to the duality of Clifford Algebra, between outer products and the meet operation.

- The "1", "T", "⊥", and "0" of linear logic correspond to 1, the unit pseudoscalar, the unit pseudoscalar again, and zero.

Going through Hestenes' book on Geometric Algebra and Girard et al's "Advances in Linear Logic", the correspondence runs so deep that I find it hard to believe the systems do not correspond.

Interestingly, the development of Geometric Algebra was developed to treat spaces and their subspaces as first-class constructs in a single algebra, with the new geometric product bringing together terms of dimensionality; while Linear Logic was invented to treat uniqueness (in a certain view, a form of dimensionality) as a first-class construct, using the "with" (do-only-once) operator to formalize the uniqueness dimension of values.

It would thus be delightful to see that these two operators, and perhaps the entire formalisms of GA and LL, contain the same substance, developed independently, as they were, almost a hundred years apart.

I would be interested in a counterexample, or references on this correspondence if it is known.

-Tim Sweeney

## 2.23 [TYPES] Better vector math using dependent types

I finally found the solution to a type theory problem that's been bugging me for years. It's a twist on the "function parameters are covariant" problem, in the special case of mathematical objects like vector spaces and matrices.

Here I will use functions from natural numbers (or an upper-bounded subset of natural numbers) to real numbers to represent vectors. You can think of such functions as fixed-length arrays of real numbers, or of floating point numbers – I will be lax with the terminology.

### The Problem

#### 2.23. [TYPES] BETTER VECTOR MATH USING DEPENDENT TYPES

When determining whether one function type (or array type, or vector type) is a subtype of another, type systems treat the function parameter type contravariantly. Conceptually, this isn't what one wants with mathematical objects like vectors.

I will use fixed-length arrays as an example below and assume classic array typing rules. However, you could see the same issue in C++ by declaring a class `vec2d {float x,y;}` and a class `vec3D: public vec2D {float z;}`, and a subclass `vec4D` containing yet another component.

Example of the problem in C-style pseudocode:

```
void Use3DVector(float[3] my3DVector) {
    ...do something here
}
void Example() {
    // Declare a 2D vector and a 4D vector.
    my2DVector b={1,2};
    my4DVector a={1,2,3,4};

    // Use the 4D vector. This succeeds because it's typesafe:
    // a 4D vector has all the components of a 3D vector and more.
    Use3DVector(my4DVector);

    // Use the 2D vector. This fails because it's not typesafe: a 2D
    // vector doesn't have certain components that a 3D vector has.
    Use3DVector(my2DVector);
}
```

In other words, traditional type systems inherently use these subtyping relationships

```
.. <: float[4] <: float[3] <: float[2] <: float[1]
```

This is the opposite of how mathematicians look at vector spaces. They see 1D space as a subspace of 2D space; 2D space as a subspace of 3D space, etc:

```
float[1] <: float[2] <: float[3] <: ..
```

## 2.23. [TYPES] BETTER VECTOR MATH USING DEPENDENT TYPES

## The Solution

Define a new type "zero" which is a subtype of "float". Type "zero" has only one instance, the value "0". By definition of subtyping, all zeros are floats, but not all floats are zeros. This is sound.

Since "zero" has only one instance, variables of type "zero" don't need to be stored; they take up no runtime storage. Since they take up no storage, you can store infinitely many of them in a fixed-sized data structure. I exploit this to represent vectors as infinite-length arrays consisting of a finite number of reals, followed by an infinite number of zeros.

Now, instead of looking at vectors in the Java style (`float[0]`, `float[1]`, etc.), we view them with dependent types, like:

```
ForAll(i nat).if i<3 then float else zero // a 3d vector
ForAll(i nat).if i<1 then float else zero // a 1d vector
```

Or conceptually, you can think of vectors as the examples below, where "... " stands for "followed by infinitely many zeros":

```
{float,float,float,zero...}
{float,zero...}
```

If you define a generic type constructor `vec(i)` in the style above, you'll find that it's exactly what mathematicians want:

```
vec(1) <: vec(2) <: vec(3) <: ...
```

You can verify this by looking at the "ForAll" expressions above, verifying that for every possible input value "i", the corresponding subtyping rules hold. For example: `{real,zero...} <: {real,real,zero...}` because (looking at the first elements) `real < :real`, and (second elements) `zero < :real`, and (subsequent elements) `zero < :zero`, etc.

## Other Benefits

### 2.23. [TYPES] BETTER VECTOR MATH USING DEPENDENT TYPES

You can then derive types from constants, i.e.  $\{2,1,0\}$  has the type of a 2D vector because the third component is 0, whose exact type is zero rather than the more general float.

You can also type strangely-shaped vector spaces like  $\{\text{float},\text{zero},\text{float},\text{zero}\dots\}$  – the space of vectors with only X and Z components.

In languages supporting dependent types and pattern matching, many mathematical rules automatically fall out of the results. For example, transforming a vector by a matrix which has one column of zeros yields a vector with the corresponding component set to zero – not just at run-time, but in the type system as well, as long as the zeros are statically known.

### Summary and Future Work

By changing our representation of vectors from fixed-length arrays of reals, to infinite-sized arrays containing finitely many reals and infinitely many zeros, vectors now obey the subtype=subspace rules mathematicians want, rather than the covariant subtyping of traditional fixed-length arrays and records.

I am currently working on extending this approach to deal with the multivectors of Geometric (Clifford) Algebra. There are many ways of doing this which don't involve any new type-theory constructs, but clearly nothing we can build out of ordinary dependent types will admit ordinary scalars, and the newly-defined vectors here, as elements of any more general multivector type.

However, I've found a new construct, sort of a "dimensionality-indexed transparent product" that gives you multivectors compatible with traditional scalars, and the vectors defined here. Better yet, it admits imaginary numbers, complex numbers, quaternions, and spinors of arbitrary dimensionality as subtypes; and it always knows the more specific derivable type of your result. This is still somewhat sketchy but I'd be glad to share it if there is any interest.

-Tim

## 2.24 [D] Persistence

If a language has complete support for introspection (discovering all fields of objects at runtime), then it's easy for users to implement arbitrary persistence algorithms, such as XML or various binary formats. So a desire for persistence can be simplified into a more general desire for complete (perhaps Java-style) introspection support.

-Tim

"Charles Hixson" <charleshixsn earthlink.net> wrote in message news:3B7BF417.5030509 earthlink.net...

*It would be desireable if D provided support for the implementation of persistant objects. Basically some way of reading back in as objects of the same type that they were when they were stored, and then using them as objects of that type.*

*As far as I can tell, this requires that "virtual" objects be allowed to be called by any function, and that they raise a "Does Not Understand" exception if the object that they actually turn out to be doesn't understand the method. Since type identification information is already being stored with each object, a large part of the necessary is already in place.*

*N.B.: I am not saying that D should implement these features. Merely that it would be desireable if the support for them were designed into the language, so that if they were implemented as libraries, they would fit in smoothly.*

## 2.25 [D] Int to string

Using "+" for string concatenation always leads to confusing ambiguities. In a language without overloading or templates, this can still work, but requires the user to sometimes perform mental gymnastics figuring out "why am I getting weird results using + in this context?"

However, if templates or overloading are present, then you run into even

worse ambiguity problems. To avoid ambiguity, you really need to have separate syntax for:

```
adding (i.e. integers).
concatenating arrays (strings just like any other case).
prepending one t to a t[].
appending one t to a t[].
```

Note that this is provable rather than speculation. :-)

-Tim

"Overlord" <peewee.telia.com> wrote in message news:9lgrsh\$2e8c\$1digitaldaemon.com...

*In D strings can be copied, compared, concatenated, and appended like*

```
str1 = str2;
if (str1 < str3) ...
func(str3 + str4);
str4 += str1;
```

*But if you want to add a integer into this, does(or will) D support it this(or in some simmilar way)?*

```
int i=10;
char[] str = "abc";

str1 = i;          // str1="10"
if (str == i) ...
func(str + i);    // abc10
str += i;         //same as above
```

## 2.26 [D] printf

Argh, no! Printf and scanf are ugly remnants of C's limitations. Printf made sense in the ANSI C days, but it and its terribly potential for run-time crashes and subtle bugs have no place in a modern language.

C#'s solution is an interesting good compromise, something like:

```
print("{1} picked up {2} {3}s", person, count, item);
```

The HUGE advantages are:

1. It is typesafe. The parameters must be convertible to strings, and their type isn't (redundently or incorrectly) specified in the format string."
2. It's easily localizable to other human languages because arguments can be reordered. This is really critical for shipping software internationally, where the order of nouns, verbs, etc., vary.

An even slightly better approach is to have a new "concatenate with formatting" operator which takes a formatting string and a variant and returns a new string. For example:

```
"{1} picked up {2} {3}s" $ person $ count$item
```

The \$ operator has the following effects:

```
"{1} picked up {2} {3}s" $ "tim" = "Tim picked up {1} {2}s"  
"Tim picked up {1} {2}s" $ 12 = "Tim picked up 12 {12}'s"  
"Tim picked up 12 {1}s" $ "marble" = "Tim picked up 12 marbles"
```

Therefore

```
"{1} picked up {2} {3}s" $ "tim" $ 12 $ "marbles" = "Tim picked up 12 marbles"
```

This is very general, and could be extended with other appropriate natural language features useful to localization such as pluralization.

-Tim

"Richard Krehbiel" <rich.kastle.com> wrote in message news:9lgkos\$251p\$1@digitaldaemon.com...

*For goodness' sake, don't take printf from C verbatim.*

*The programmer himself must match the format specifier to the data type. The programmer himself must align the number of the format specifiers to number of arguments. The programmer himself must align the positions of the format specifiers the positions of arguments.*

*Make a printf where the format specifier is adjacent to the variable.*

*I have a function that works this way:*

```
my_printf("a=%d", (int)a, " b=%d", (int)b, (char *)NULL);
```

*i.e. it takes a format string containing \*one\* argument; the argument(s) following are for that format specifier; multiple pairs of format/argument can be supplied.*

*Perhaps in D, the end of the variable argument list to Dprintf can be rather than having the programmer be diligent to add the NULL at the end every time. Actually, if you add Real Macros, printf can be a macro which knows the number and data types of the arguments:*

```
Dprintf("a=", a, " b=", b, "\n");
```

*– Richard Krehbiel, Arlington, VA, USA rich.kastle.com (work) or krehbiel3.home.com (personal)*

## 2.27 [D] \*sigh\*

*I personally miss the preprocessor when I use Java.*

Agreed, but only because of Java's limitations (no templates, etc.), and not because I \*want\* to use macros.

*And templates! Yes they are complex stuff and I wish they were easier but... they allow you to do things that you would otherwise be unable to do, or have to pay dearly for.*

Agreed 100%! I would never consider writing a large-scale program in a language without template-style functionality. It's really astounding how much simplification and generality one can attain with templates – not just talking about the canonical (and IMO poor) example of "container classes", but for complex related type hierarchies, parser tools, math structures, etc.

The stigma of templates being useful just for container classes and a few other isolated cases (which a language could well implement as a special feature) is really unfortunate. I suppose that's because container classes are the next obvious step for a C programmer. But if you spend a few weeks experimenting with languages like Haskell (not useful for practical programming, but a GREAT thing to learn), you come back to C++ templates with a completely new perspective and start writing code in a much more compact and general way!

(Here I'm advocating template-style functionality – I actually think the C++ syntax for templates is pretty kludgy, but that's a minor issue in the grand scheme of things.)

*Unfortunately, right now in C++, it is difficult and unintuitive to do, but it is possible (there's a nice book that discusses some of these things, and you'll be amazed at what they can get templates to do - Generative Programming by Krzysztof Czarnecki, and Ulrich W. Eisenecker).*

Agreed 101%!

-Tim

## 2.28 [D] Operator overloading

As someone who writes tons of vector math code in production applications, my feeling is that a language which doesn't support operator overloading and template-style programming is really going to be really painful for modern math-intensive applications, such as games, modelling programs, etc.

I'm not necessarily advocating C++'s approach (operator overloading can be confusing, such as using an overloaded "<<" operator to represent both bit shifting and character output!)

Haskell-style "typeclasses" are an interesting replacement for general operator overloading. This approach encapsulates the idea that a given operator like "+" or "\*" should have the same notional meaning everywhere, even when operating on different data types.

A simple example is that "+" and "\*" would belong to the "numeric" typeclass. If you create a new numeric typeclass of your own (for example, a "bigint" class), then you declare your class to belong to the "numeric" typeclass, and implement those specific operators. However, you wouldn't be able to create a "+" operator for a non-numeric class.

This can map to popular languages easily by having your class implement an abstract interface (aka typeclass) and provide an appropriate static or friend function for those operators.

However, this approach doesn't extend well to more general situations. For example, say you define a template like `vector<float,4>` (a mathematical 4-component vector), one wants the ability to multiply those vectors by scalars using "\*", which requires the more general form of overloading.

-Tim

"John Fletcher" <J.P.Fletcher@aston.ac.uk> wrote in message news:3B78E154.BC62E006@aston.ac.uk...

*Quote from the specification for D:*

*Operator overloading. The only practical applications for operator overloading seem to be implementing a complex floating point type, a string class, and smart pointers. D provides the first two natively, smart pointers are irrelevant in a garbage collected language.*

*Another quote:*

*D has many features to directly support features needed by numerics programmers, like direct support for the complex data type and defined behavior for NaN's and infinities.*

*Comment:*

*For numerical computing it is convenient to define classes e.g. vectors, matrices and other entities beyond complex numbers, such as quaternions. For this overloading of operators such as +, -, +=, etc means that top level code can be easily written and readable.*

*John Fletcher*

## 2.29 [D] Macros

*Okay, why is it that everybody thinks the C preprocessor is terrible and needs to be avoided?*

Because it's terrible and needs to be avoided, of course!

*C++ only has it for compatability; Java and C# and D don't have one. I just don't get this one.*

Historically (beginning with C), macro preprocessors have served as a kludge to compensate for a lack of language power and optimizer ability. Macros were good and necessary in the C days, because you could often come pretty close to C++ style template programming if you were clever enough, using macros for casts and other tricks. You could also use them to construct optimized code (i.e. by defining the body of a loop in a macro, and unrolling the loop 100 times with the macro).

With a modern language, these kludges should not be necessary. If there is any particular case where you feel you could express a program more cleanly by using macros, then I recommend looking at that as a language or optimizer flaw to be fixed, and not a need for macros.

Unreal (the game) is probably a good production-code example. It's around 250,000 lines of code and uses macros for the following:

1. To expose metaclass information (i.e. class names, default constructors that a serializer can call) – like MFC's techniques. All of this code

would be unnecessary if the language supported classes as first-class objects (i.e. you can pass around a `classref*` which "represent" the class and exposes its static functions), static virtual functions, and static constructors. 2. To comment out large blocks of code. Would be unnecessary if `/*...*/` comments could be nested. 3. To implement debug-specific code. This is actually unnecessary, a bad old habit. We would be just as well off having a global constant `debug=0` or `1`, and having `if(debug) {...}` instead of `#if debug`. 4. To implement platform-specific headers. Only necessary because headers are necessary. 5. To perform template-style tricks. If the language has a great facility for type dependency (whether like C++ templates, or more general like Haskell), all of these things would be unnecessary. Even C++ templates aren't complete enough, i.e. there are no template typedefs (true type synonyms), and most production compilers have bizarre template bugs limiting what you can do.

I'm 100% sure the Unreal code would be simpler and cleaner if the language supported the above and all preprocessor support were eliminated.

-Tim

## 2.30 [D] Getters and setters

*In D, get'ers and set'ers take advantage of the idea that an lvalue is a set'er, and an rvalue is a get'er:*

```
class Abc
{
    int myprop;
    void property(int newproperty) { myprop = newproperty; } // set'er
    int property() { return myprop; } // get'er
}
```

*which is used as:*

```
Abc a;
a.property = 3; // equivalent to a.property(3)
int x = a.property; // equivalent to int x = a.property()
```

Why??!? :-) In current reasonable languages "a.x" is a variable access and "a.f(x)" is a function call.

Why complicate this so that "a.x" could either be a function call or a variable access, depending on its (possibly very complicated) context?

Also, this seems to create ambiguity (or context-specific special-casing of semantics) with function pointers. Given a reference to a function p taking a parameter, p=a could either mean calling it with a parameter of a, or initializing it to a.

I know Bertrand Meyer advocated this approach in Eiffel, but this has been shown to be one of several areas (along with, i.e. covariant typing on function parameters) where Eiffel's type system is unsafe and problematic.

*Thus, in D you can treat a property like it was a simple field name. A property can start out actually being a simple field name, but if later it becomes necessary to make getting and setting it function calls, no code needs to be modified other than the class definition.*

That breaks with open-world modules. For example, given a module declaring a class like:

```
class c
{
    int a;
};
```

How can you subclass it later on in another module like:

```
class d: public c
{
    int a() {...}
};
```

The similar question comes up with binary-compatible evolution of modules (i.e. Java's list of rules, "You can do the following things to evolve a class without breaking existing precompiled modules that depend on it". That's an essential thing that's vital to writing real-world programs, such as applications supporting plug-ins. The only sound solution to this is that every variable access in a program has to translate to a virtual function call, which obviously isn't reasonable.

-Tim

## 2.31 [D] Your feelings

*Frequently in the D news group you reject ideas because they don't "feel right". Respectfully submit that this is not a basis for informed decision. printf doesn't "feel right" to me and never did.*

Language syntax is a creative work, influenced by history and aesthetics. Making a language "feel right" is vital to its success!

*If D cannot use the STL (or a wrapped version of the STL), not many people will be using it.*

Baloney! STL is something only a C++ programmer could appreciate.

Witness that when STL is translated to a language with a more flexible type system (such as Haskell), it basically "disappears" into much simpler native language functionality – and you realize that half of STL consists of workarounds for C++ limitations, while the other half consists of workarounds for STL's limitations.

STL-like techniques have no place whatsoever in a cleanly designed language.

-Tim

## 2.32 [comp.lang.c] Re: Parity of a number

Aw, c'mon guys. For n bits, you only need  $\log_2(n)$  xors and shifts.

```
uint16_t d;
uint16_t t0,t1,t2,t3;
int parity;

t0=d^(d>>8);
t1=d^(d>>4);
t2=d^(d>>2);
t3=d^(d>>1);
parity=t3&1;
```

-Tim

## 2.33 [comp.lang.c] Re: Parity of a number

*His code is wrong but his statement is right, and your statement about a worst-case complexity of  $\Theta(n)$  for the parity of an n-bit number is wrong. Consider the following code:*

```
int uint16_parity( uint16 x ) { x = x ^ (x >> 8); x = x ^ (x >> 4);
x = x ^ (x >> 2); x = x ^ (x >> 1); return x & 1; }
```

Gunnar,

Thanks for the correction. I should've known better than to post a piece of code without testing it. :-)

-Tim

## 2.34 [sci.math] Constructing the integers from set theory without equivalence classes

There exist many simple ways to construct the natural numbers from set theory, such as:

$$0 = \{\} \quad n+1 = \{n, \{n\}\}$$

Then one typically constructs the integers from equivalence classes of pairs of natural numbers, such that:

$$\{a_0, \{a_1\}\} = \{b_0, \{b_1\}\} \text{ iff } a_0 + b_1 = a_1 + b_0$$

Is there a simple way to construct the integers directly from set theory, without making use of equivalence classes? In other words, I'm looking for a simple representation of the integers such that each integer is represented by a unique set, rather than an equivalence class of sets.

As a bonus it would be nice if the representation admitted a straightforward definition of addition, subtraction and multiplication.

Thanks,

Tim Sweeney

## 2.35 [sci.fractals] The nth iteration of $f(x) = x^2 + c$

Here I will use the notation  $f^{-1}(x)$  for the inverse function of  $f(x)$ , and in general  $f^n(x)$  for the nth iteration of  $f(x)$ . For example,  $f^3(x) = f(f(f(x)))$ .

Some obvious identities involving iterated functions are:

$$\text{if } f(x) = x + c \text{ then } f^n(x) = x + nc \quad \text{if } f(x) = x * c \text{ then } f^n(x) = x * c^n$$

A few more identities can be derived in closed form, such as  $f(x) = a * x + b$ . However, deriving a general formula for  $f^n(x)$  turns out to be remarkably hard, even for simple  $f$ .

Now, given  $f(x) = g(h(g^{-1}(x)))$ , it's easy to see that  $f(f(x)) = g(h(h(g^{-1}(x))))$ .

And more generally,  $f^n(x) = g(h^n(g^{-1}(x)))$ .

This identity is useful for determining the  $n$ th iteration of a function  $f(x)$ , when we can represent  $f(x)$  as a composition of the functions  $g$  and  $h$ , where  $h$  is a function simple enough to determine  $h^n(x)$  directly.

For example, we know that

$$2 \cos(x)^2 - 1 = \cos(2x)$$

Replacing  $x$  with  $\arccos(x)$ , we have

$$2x^2 - 1 = \cos(2 \arccos(x))$$

Now,  $\cos(2 \arccos(x))$  is of the form  $g(h(g^{-1}(x)))$  where  $g(x) = \cos(x)$  and  $h(x) = 2x$ . Therefore, we derive

$$\text{if } f(x) = 2x^2 - 1 \text{ then } f^n(x) = \cos(2^n \arccos(x))$$

When I derived this result 13 years ago, it went a great way towards understanding the behaviour of chaotic dynamic systems: in iterating  $2x^2 - 1$  many times, one quickly notices that small perturbations of the input value have a dramatic effect on later results. The mechanism for this becomes much less mysterious when you consider the equation  $\cos(2^n \arccos(x))$  as  $n$  grows large: the function is zooming through oscillations of the cosine function at an exponential rate.

With some simple scaling that doesn't affect the end results, one can transform these equations to

$$f(x) = x^2 - 1/2$$

Which just happens to be a point on the Mandelbrot set (defined by iterations of  $f(x) = x^2 + c$ ) where  $c = -1/2 + 0i$ .

So, I have a few questions related to this topic:

1. Are there any studies or research papers on the general study of determining  $f^n(x)$  for various functions?
2. Is a closed-form solution known for  $f^n(x)$  given  $f(x) = x^2 + c$ ?
3. Are there any studies or general results on  $f^n(x)$  where  $n$  is:

- Rational. Such a notation is clearly meaningful, i.e.  $f^{2/3}(x)$  is the function  $g(x)$  such that  $f(f(x))=g(g(x))$ .

- Real, i.e.  $f^{\pi}(x)$ . The iterative meaning of such a notation isn't obvious but from the obvious example of  $f(x)=x+c$  implying that  $f^n(x)=x+n*c$  it seems that such an extension could be defined.

Thanks,

Tim Sweeney

## 2.36 [sci.math] Re: Hestenes' Geometric Algebra

*I have downloaded a lot of papers by David Hestenes and others on geometric algebra and am reading the book Clifford Algebra to Geometric Calculus by Hestenes, but I am still looking for some practical examples of calculations done using this system of geometric algebra. Does anyone know of any examples?*

Hello,

In writing 3D graphics code in C++, I've used geometric algebra to represent points (3D vectors), planes (3D bivectors), and quaternions (3D spinors, the sum of a bivector and a scalar). This approach works quite well, though I haven't yet found any fundamental improvements in using geometric algebra over more traditional matrix approaches.

Hestenes presets the basis-free nature of geometric algebra as one of its biggest strengths, but if you're a computer guy, the first thing you'll do in writing a geometric algebra library is to make the basis explicit.

-Tim

## 2.37 [comp.lang.functional] Extending Church Numerals to integers, rationals

Does anybody know of work on extensions of Church Numerals beyond the natural numbers?

Church Numerals are a convenient way of encoding natural numbers in the lambda calculus. A Church Numeral  $N$  is a function that, given a function  $F$  as input, returns  $F$  iterated  $N$  times. In other words:

$$\begin{aligned} 0 &= \lambda(f).\lambda(x).x \\ 1 &= \lambda(f).\lambda(x).f(x) \\ 2 &= \lambda(f).\lambda(x).f(f(x)) \\ n+1 &= \lambda(f).\lambda(x).f(n(f)(x)) \end{aligned}$$

Church Numerals are interesting because they admit a very simple representation of addition, multiplication, and exponentiation:

$$a+b = \lambda(f).a(f)(b(f)) \quad a*b = \lambda(f).a(b(f)) \quad a^b = \lambda(f).\lambda(g).g(f)$$

In this spirit, we can write computationally valid (though perhaps mathematically odd-sounding) expressions such as:

$$\sin * \cos(x) = \sin(\cos(x)) \quad \sin^3(x) = \sin(\sin(\sin(x)))$$

One can imagine extending Church Numerals (as entities of the lambda calculus) beyond the natural numbers in an analogous way to extending the natural numbers to the integers, rationals, or reals: just as "2" is a function that iterates its argument twice, "-1" is a function that returns the inverse of its argument.

I'm intentionally oversimplifying, of course: such an "inverter" function only makes sense when its argument is a function with a valid inverse, but that does not necessarily invalidate the concept. Such a condition of invertability could perhaps be expressed in a sufficiently powerful dependent-typed lambda calculus.

I'm interested in any pointers to research along these lines because this seems like it may be a fruitful way of extending well-known mathematical concepts (such as addition, multiplication, and exponentiation) into functional concepts, in a well-defined way.

-Tim

## 2.38 [comp.lang.functional] Re: Extending Church Numerals to integers, rationals

I wasn't expecting someone to mention ZZT here! :-) Once in a while I take a short break from game programming and work on an interesting side-project. Currently I'm fascinated with the applications of type theory, and am experimenting with extensions that increase expressiveness at the expense of decidable typechecking and possibly even reduction.

Regarding Church Numerals, another thought along these lines is to extend the lambda calculus with a new primitive "-1", not itself definable as a finite lambda term, and define a new (possibly incomplete) reduction rule for it which captures the essence of a negative church numeral: that for all terms  $f$  and  $x$ ,  $f(-1(f)(x)) = -1(f)(f(x)) = \text{id}$ .

It was mentioned that -1 is the fixed point of  $y = y + y + 1$ , but that leads to an infinite expansion that does not appear to be computationally useful.

It is interesting to restate a problem such as "y such that  $f(y) = x$ ", as  $-1(f)(x)$ . Not that this causes a solution to magically appear! But it does move the problem from solving an equation to evaluating an expression in a lambda calculus augmented with the primitive "-1".

The notion of negative Church Numeral terms makes varying degrees of sense in different calculi. It is most interesting in a logic programming framework supporting terms with multiple "potential" values such as McAllester's Ontic: <http://www.autoreason.com/>. In this sort of framework, it seems that every function has a unique and perfectly sensible (though possibly multivalued, and not necessarily computable) inverse.

Regarding exponentiation (" $a^b = \text{lambda}(f).\text{lambda}(g).g(f)$ "), this agrees with convention for all natural numbers, except for the somewhat questionable case of  $0^0 = 0$ .

[Disclaimer: I'm presenting this experimentally and have not attempted to prove the soundness of the proposed extensions.]

## 2.39 [microsoft.public.dotnet.languages.csharp] C# for game development?

Does Microsoft take the possibility of using C# for game development seriously? It seems like it could be considered either as a gameplay scripting language or as a full-blown development language once DirectX9 ships with full .NET support.

The language seems very capable in most regards and has many attributes of an excellent game scripting language, but its garbage collection pauses seem to be a showstopper flaw. In prototype code that does an amount of realtime object creation that is reasonable for a current generation game, i.e. around 100 object allocations per frame at 60 fps, I'm seeing pauses of 2-3 seconds that occur once every 5-10 minutes.

Is this a feature or a bug?

Obviously one can't ship an action game that hits a 5 second pause every 5-10 minutes!

Note that in this scenario the amortized time going to garbage collection (under 3% of execution time) is totally acceptable. What's unacceptable is that it all happens at once and makes the application appear to be locked up. In a realtime app such as a game that attempts to run at 60 fps, it is necessary to establish a realtime expectation on garbage collection overhead (not a hard guarantee, but an expectation that we can count on being broken only very infrequently), such as "the garbage collector will never use more than 1 msec within any 10 msec period of time".

Also, out of curiosity, is this the same issue that causes the Visual C++ IDE to lock up for 5 seconds every 5-10 minutes? (I was assuming this was related to the non-Microsoft Perforce source control integration we use, but after experimenting with C# I wonder if this is an unavoidable aspect of managed code).

The .NET framework is awesome and I really hope this isn't a fundamental flaw of managed .NET code.

-Tim

## 2.40 [microsoft.public.dotnet.languages.csharp] Concatenate two arrays?

Is there a reasonable way to concatenate two arrays in C#? As far as I can see:

1. There is no built-in operator to concatenate arrays.
2. There is no way of writing a general array concatenation function because C# doesn't support templates (as one would use in C++) and doesn't support low-level hacking to work around the need for casts (as one would use in C).

So, every time I want to concatenate two arrays, I have to create a new array of the proper length, then copy both arrays' data into the new array - something that takes 3 big complex lines of code, when I really want to just say "a+b".

Any thoughts? (Please pardon me if I'm missing something obvious!)

-Tim

## 2.41 [comp.lang.functional] Re: Interpolating plus, multiplication etc

$F(A,B,0) = A+B$   $F(A,B,1) = A*B$   $F(A,B,2) = A^B$  For instance  $F(3, 4, 3) = 3^{(3^{(3^3)})}$

*So F grows rather rapidly with 'i'.. What I would like to do is to interpolate F so that 'i' may be any real number, which would produce a function where addition, multiplication etc are only a special case. I think this is very fascinating, but I really have NO IDEA of how to do it.*

This is a fascinating question, but I too have been unable to find any research of significance on the topic. It seems quite curious that such a fundamental question remains not only unanswered but also practically unasked.

Thoughts:

- Such a function of  $a, b, i$  isn't expressible as a Taylor series, because it grows arbitrarily fast.
- We shouldn't expect such a function to be commutative or associative for arbitrary  $i$ .
- It is not clear whether such a function would be continuous.
- It is not clear that mainstream mathematical tools are suitable for answering this question. In fact, given the lack of research into such topics, it is likely not.

Here is another interesting way of asking the question, which I pose recreationally because it hasn't been studied rigorously.

Rephrase your problem in the lambda calculus by defining  $f'$  as  $(\text{lam } i)(\text{lam } a)(\text{lam } b)f(a, b, i)$ . And then  $f(a, b, i)$  is just  $i'(f)(a)(b)$ , where  $i'$  is the generalized Church numeral corresponding to the real number  $i$ .

What is a Church numeral? It's a way of encoding numbers as functions, such that:

$$0 = (\text{lam } f)(\text{lam } x)x \quad 1 = (\text{lam } f)(\text{lam } x)f(x) \quad 2 = (\text{lam } f)(\text{lam } x)f(f(x)) \dots$$

What is a generalized Church numeral? It's their extension to the reals:

$-1(f)$  = "the" inverse of the "function"  $f$ .  $1/2(f)$  = "the" "function"  $g$  such that for all  $x$ ,  $g(g(x))=f(x)$

Where "the" and "function" are oversimplifications (in general, such objects are not one-to-one and are thus more general relations than functions, and in general they are not unique, so formally we might talk about equivalence classes of relations satisfying the Church numeral criteria.)

How does one compute with generalized Church numerals? This is an open question which, to my knowledge has not been studied. However, I think this does represent at least meager progress, in the same way that writing  $i=\text{sqrt}(-1)$  enabled the study of complex numbers before their properties were fully understood. To study something, one must first acknowledge that the thing might exist.

-Tim

## 2.42 [TYPES] Re: Type inference and related research...

Hello,

If your interest is directed towards type systems of real-world programming languages, focusing on type inference is not necessarily the most productive direction to take. As I see it, the subject mostly dead-ends around the point of sophistication of the Haskell type system, after which it becomes undecidable or at least unwieldy. This does not seem like a feature that will make it into future programming languages that are widely used. Though, the underlying concept of unification is very powerful and seem more promising – if unification is your primary interest, then it could be very useful to explore it in other, more practical, situations than type-inference algorithms.

Here are two type-theoretic subjects that I feel hold the most promise towards improving future type systems:

1. The representation of "very dependent types", as originated by Jason Hickey. See <http://www.cs.caltech.edu/~jyh/papers/fool3/default.html> for the original paper. In earlier type systems, terms could reference (by symbolic binding or de Bruijn index) the 'parameter' part of outer lambda expressions. Very dependent types also expose the 'function' part of outer lambda expressions, but in a more general way than the recursive mu binder: they preserve partially-known information.

Thus you can use very dependent types to form arbitrarily complex dependent-typed records with recursive dependencies. This provides a framework that seems an ideal framework for representing object-oriented concepts such as self-type dependence (without resorting to a new binder as in Cardelli's object calculi), as well as uniformly representing dependent mathematic structures such as groups or rings. In my experience, very dependent types have proven very easy to work with and well-known concepts such as Cardelli's explicit substitutions approach, and Cardelli's

subtyping-of-recursive-types are easily extensible to very dependent types.

As far as I'm aware, there is only one paper on this topic and the value of this research topic hasn't been widely recognized yet.

2. Type systems that combine non-determinism (at least at the type level) with type-forming and type-specification operators. The breakthrough paper on the topic is David McAllester's description of the Ontic language: <http://ttic.uchicago.edu/~dmcallester/ontic-spec.ps>. This also appears to be the only paper on the topic and the value of the research hasn't been widely recognized yet.

The notion of non-determinism in Ontic isn't the scary or impractical kind that is associated (by practical programmers) with languages like Prolog; here it can be used purely as a type-system tool that enables you to express such concepts as "the type containing the elements 1, 2, and 3"; "an element whose value is either 1, 2, or 3", "a subtype of the type of natural numbers", "the type of all subtypes of natural numbers", etc. The Ontic type system provides a direct and intuitive representation of concepts which are either difficult or indirect and circuitous to express in more traditional dependent-type systems.

One of the neat things of Ontic is that the idea of a typechecker assigning type judgements to each term context in the form "a:T" is replaced by the more general idea of assigning a (hopefully finitary) set of possible values to each term. This makes it easier to deal with complex subtyping situations such as f-bounds and recursive types.

Please note that I am not a type theorist, but a practical programmer who follows type theory research in search of ideas that may contribute towards future practical programming languages, so my advice here is biased in a direction that others are likely to disagree with.

-Tim

## 2.43 [sci.math] Extending GCD,MOD operations to rationals

Hello, A common mathematical definition of the greatest common denominator  $\text{GCD}(a,b)$  expresses the function in terms of the prime factors of the arguments:  $\text{GCD}(2^a 3^{a_1} 5^{a_2} \dots, a^b 3^{b_1} 5^{b_2} \dots)$  is  $2^{\min(a,b)} 3^{\min(a_1,b_1)} 5^{\min(a_2,b_2)} \dots$ . This definition extends easily to rational numbers, where the exponents of the prime factors may be negative. So we have, for example,  $\text{GCD}(2/3,5)=1/3$ .

The standard GCD algorithm on natural numbers (in C++ syntax) is:

```
int gcd(int a,int b) {return a==0? abs(b): b==0? abs(a): gcd(b,a%b);}
```

Where "a%b" means "the remainder obtained when dividing a by b".

In an attempt to be clever, I've extended the remainder operation to rationals by defining  $(a/b)\%(c/d)$  to be  $(ad\%bc)/bd$  where a,b,c,d are integers. This appears to extend the above GCD algorithm to rationals in the expected way. Any comments on whether this is sound?

Also, given possibly negative integers a,b, is there a standard convention on the sign of  $\text{gcd}(a,b)$ ? I'm assuming it should be positive (it's the \*greatest\* common denominator after all), but don't want to miss out on any potential generality that could be gained.

Finally, is there an extension of the gcd operation to reals that brings some meaning to terms like  $\text{gcd}(e,\pi)$ ? Obviously in this case one would have to reason with something besides prime factorizations.

-Tim Sweeney

## 2.44 [TYPES] Classifying the cardinalities of types

I'm developing a programming language similar in its type structure to McAllester's Ontic, where types and values are represented in a unified, dependent-typed framework. One of the major problems of the compiler is to characterize the cardinalities of various types in programs, and I'm

posting this to informally share my results, and see if anyone has references to similar work. Here goes:

Every type  $T$  can be thought of as having a cardinality,  $\text{Card}(T)$  characterizing the number of unique elements of that type  $T$ . The cardinality of the empty type is 0, the cardinality of the unit type is 1, the cardinality of the boolean type is 2, the cardinality of the natural numbers is a countably infinite cardinal.

In this language, we can say:

- "1" is a single value, so its cardinality 1.
- "1|2|3" is either 1, 2, or 3. It has 3 potential values, so its cardinality is 3.
- "a-natural-number" can potentially be any natural number, so its cardinality is countably infinite.
- Given a variable "x" representing an unknown natural number, "x|3" reduces to the single value "3" if  $x=3$ , or the two distinct potential values "x" and "3" if  $x \neq 3$ . In this case, we have to say its cardinality is 1|2. Thus to each term the compiler assigns not a single cardinality, but a set of potential cardinalities.

Characterizing the precise set of potential cardinalities of a term is a hard problem, especially as the complexity of the type system grows. However, conservatively approximating the potential cardinalities of an expression is tractable, and this is the course I've pursued.

I draw potential cardinalities from the set  $\{0,1,m\}$  where "m" stands for "many". Thus we can precisely characterize the cardinality of the empty type as 0 and the unit type as 1, while the cardinality of 1|2|3 and a-natural-number are both conservatively approximated as "m". The expression "x|3" above has cardinality 1|m.

This affects the language design in the following way. Each type-theoretic operator (such as "intersection of types", "union of types", "product types", "disjoint union of types" has a table characterizing the potential cardinality of the result as a function of the cardinalities of the input. For example, in  $\{0,1,m\}$  approximation, in a language with only atomic values and not functions, we can describe the union operation with:

#### 2.44. [TYPES] CLASSIFYING THE CARDINALITIES OF TYPES

$\{0\} \{1\} \{m\} \{1\} \{1|m\} \{m\} \{m\} \{m\} \{m\}$

In other words, the union of 0 with an element cardinality  $x$  is itself  $x$ , while the union of two elements of cardinality 1 can either be of cardinality 1 (if the elements are equal) or  $m$  (if unequal).

Everything I've said so far can be applied to set theory as well as type theory.

Now, the presence of functions in the type system complicates the cardinality characterization considerably, and becomes dependent on the intricate details of the type theory.

If the full variance of functions is built into the type theory, then every function of the form  $a \rightarrow b$  is a subtype of  $c \rightarrow d$  if  $b < :c$  and  $c < :a$ . In a type system without a universal element ( $u$  such that every type in the type theory is a subtype of  $u$ ), every function is of cardinality "m", because every function has infinitely many subtypes. In the presence of a universal element, functions from the universal element to elements of cardinality 1 is itself of cardinality 1 because there are no subelements.

Since practical functions like  $\lambda(x:a\text{-natural-number}).x$  have infinitely many subelements, they receive cardinality "m". This isn't completely satisfactory, because my language requires that programs be observationally deterministic – "3" is a valid program, while "3|4" is not (though 3|4 may be used as an intermediate term in part of the program, for example as the domain of a function). Unfortunately classifying a function as cardinality "m" (which is necessary for the cardinality to be correct) prevented programs from being functions.

The realization that saved the day is that we can make a distinction between items that are observably multivalued (like 1|2) and items that are observably single-valued (like that function above) but, because of the nature of function variance, are inhabited by multiple subelements. To make the distinction, I draw cardinalities from the set  $\{0,1,m,1f,mf\}$  where  $\{1f,mf\}$  categorize sets of functions and  $\{1,m\}$  categorize sets of atomic values. Examples:

"1" has cardinality 1 "1|2" has cardinality m " $\lambda(x:a\text{-natural-number}).x$ " has cardinality 1f. " $\lambda(x:a\text{-natural-number}).x|x+1$ " has cardinality

mf. " $1|2|\lambda(x:a\text{-natural-number}).x$ " has cardinality  $m|om$ .

I have lookup tables for all of the basic type operations over the cardinality set  $\{0,1,m,lf,mf\}$  and everything behaves as expected. I can think of many extensions to the cardinality set of varying degree of usefulness but  $\{0,1,m,lf,mf\}$  seems sufficient for my purposes.

I'm interested if anybody has any comments or references to similar work. So far the only discussion I've found of cardinality is in the paper on Mercury ([http://www.cs.mu.oz.au/research/mercury/information/doc-latest/reference\\_manual](http://www.cs.mu.oz.au/research/mercury/information/doc-latest/reference_manual)) where there is a lattice of "determinism categories", and that only scratches the surface.

-Tim

## 2.45 [TYPES] Re: Classifying the cardinalities of types

Hi David,

Here are the practical reasons:

1. Since my programming language is aimed at practical applications, I require that all complete programs be deterministic – in other words, any non-determinism (multivaluedness or zerovaluedness) is limited to internal computations, and not the observable results of running the program. This requires verifying that the program's type is of cardinality 1.
2. Like Ontic, I use nondeterminism to express the idea that a term has more than one potential value. Thus  $\lambda(x\ 1|2|3|4).x+7$  is a function whose domain is the range of integers from 1-4. This particular expression can't reduce any further: we can't evaluate " $x+7$ " because we don't know what value  $x$  will take on. However, we can reduce the function  $\lambda(x\ 3).x+7$  to  $\lambda(x\ 3).10$ , because we know that " $x$ " can only take on the value 3. In general, whenever reading a function parameter, or a "recursive self-value", I need to examine the cardinality in order to determine whether a reference (formally, a de Bruijn index) can be

released or whether it needs to remain there in the normal-form. This is why the potential cardinality set  $\{0,1,m\}$  works well for my application: all the language rules intimately care about is whether a term is uninhabited, single-valued, or multivalued. There is no compiler logic special-cased, for example, for cardinality-2 elements.

By "recursive self-value", I mean an expression like  $\{x\ 3|5|7,y\ x+3\}$ . I represent such things as arrays with a special kind of de Bruijn index referring back to the enclosing function, rather than its parameter. This turns out to be a very expressive notation, allowing object-oriented data structures a la James Hickey's "Very Dependent Types" – a type can not only depend on function parameters, but also on recursive values that aren't precisely known at specification time. For example  $\{t:\text{type},n:\text{nat},a[n]:t\}$  is an array containing a type  $t$ , a natural number  $n$ , and an array of  $n$  elements of type  $t$ . It's of cardinality  $m$ , but unifies with (for example)  $\{\text{char},7\}$ ,

3. The language takes the Curry-Howard isomorphism literally, and is the first language I know of that attempts this. (Whether this is a good idea remains to be proven). In it, "false" represents the 0-element type. For every type "t" (which is inherently a single-valued thing), the term ":t" means "the set of potential values of that type". Thus  $:\text{false}$  is of cardinality 0,  $:\text{unit}$  is cardinality 1,  $:\text{int}$  is of cardinality  $m$  (multivalued), etc.

My conditional expressions "if a then b else c" actually branch based on the cardinality of the condition. So there is no special boolean type along the lines of Haskell's "type Bool = True | False". If the cardinality is known at compile-time, the conditional expression reduces to one of its branches. Otherwise (i.e. if the cardinality is  $0|1|m$ ), it remains in normal form.

This brings up another topic of having to determine, at compile-time, for expressions whose cardinality is multivalued, whether the runtime is capable of reducing it to a known cardinality – something that doesn't have to be done perfectly; any conservative approximation is fine. In other words, if you say "if {fermat's last theorem} then 2 else 3", the compiler gives you an error indicating that the condition is undecidable by the compiler.

## 2.46 [comp.lang.functional] Re: Y combinator

*I have a question: At least in my implementation, the base function is transformed to take an extra argument which of course will be the function itself. This is fine for when the function is applied solely within the body of the function itself because I'll transform every use of the function to pass in the extra argument. The problem is this: What should happen if the function is passed to some other function not expecting the extra argument? What if the recursive function returns itself?*

This should all work automatically if you're managing your environments/closures properly.

If you are looking into a purely runtime solution, look for information on closures (a combination of a function, and an "outer environment" - a binding of values to all of the variables in outer function contexts). In an ordinary language, at runtime, all variables in lambdas "outside" the current lambda are guaranteed to have well-defined values when it comes time to apply the current lambda.

If you're looking for a compile-time solution, check out Luca Cardelli's paper on "Explicit Substitutions". This is a compile-time mechanism for keeping track of bindings, and scales well to complex situations of functions calling functions recursively with functions as parameters. See <http://www.pps.jussieu.fr/~curien/ExplicitSub.pdf>. If this is too big a leap (the presentation is very technical) do a Google search for "de Bruijn indices" for background.

Personally, I find it easiest to represent recursive functions without the Y combinator approach (i.e. without passing a function "to itself"). Instead, I use two distinct sets of de Bruijn indices: "parameter indices" referring to the parameter passed to the function, and "self indices" referring to the function itself. This approach is a bit more compact, but more importantly, there tend to be different compile-time evaluation semantics associated with self indices and with parameter indices: you can't try to eagerly resolve self indices or else you run into infinite loops. With the Y-combinator approach it's not obvious whether a given parameter may

be so recursive. This probably doesn't matter in simple evaluations situations, but becomes important in recursive dependent type systems.

-Tim

## **2.47 [slashdot] We need 64-bit TODAY**

Intel's claims are wholly out of touch with reality.

On a daily basis we're running into the Windows 2GB barrier with our next-generation content development and preprocessing tools.

If cost-effective, backwards-compatible 64-bit CPU's were available today, we'd buy them today. We need them today. It looks like we'll get them in April.

Any claim that "4GB is enough" or that address windowing extensions are a viable solution are just plain nuts. Do people really think programmers will re-adopt early 1990's bank-swapping technology?

Many of these upcoming Opteron motherboards have 16 DIMM slots; you can fill them with 8GB of RAM for \$800 at today's pricewatch.com prices. This platform is going to be a godsend for anybody running serious workstation apps. It will beat other 64-bit workstation platforms (SPARC/PA-RISC/Itanium) in price/performance by a factor of 4X or more. The days of \$4000 workstation and server CPU's are over, and those of \$1000 CPU's are numbered.

Regarding this "far off" application compatibility, we've been running the 64-bit SuSE Linux distribution on Hammer for over 3 months. We're going to ship the 64-bit version of UT2003 at or before the consumer Athlon64 launch. And our next-generation engine won't just support 64-bit, but will basically REQUIRE it on the content-authoring side.

We tell Intel this all the time, begging and pleading for a cost-effective 64-bit desktop solution. Intel should be listening to customers and taking the leadership role on the 64-bit desktop transition, not making these ridiculous "end of the decade" statements to the press.

If the aim of this PR strategy is to protect the non-existent market for \$4000 Itaniums from the soon-to-be massive market for cost-effective desktop 64-bit, it will fail very quickly.

-Tim Sweeney, Epic Games

## **2.48 [comp.lang.functional] Re: Why isn't Haskell mainstream?—A newbie's view**

To be realistic, there isn't lots of example code available in Haskell for mainstream tasks like file I/O, networking, and user interface programming because Haskell isn't a good language for doing these kinds of things. It's far easier to perform such tasks in imperative object-oriented languages like C++, Java, C#, and Python, so that's what people do. In Haskell, you have to resort to monads which lead to explicit plumbing of state that is far more difficult to understand than the sequential imperative model.

Most of the example code for Haskell deals with sorting, searching, functional parsers, lambda evaluators, and other similar functional recursive algorithms because Haskell is a great language for doing that sort of thing, clearly more natural than mainstream languages like C++ and Java.

So programmers flock to mainstream languages for performing primarily mainstream tasks, and to functional-recursive languages for performing primarily functional-recursive tasks, and the example code for all of these languages just tends to reflect what the languages are best at.

## **2.49 [comp.lang.functional] Re: curried function calling**

Alan,

The way you're describing this code reflects the LISP view of program-

ming, where there is such a thing as a "function with a parameter attached to it", and that "evaluating the function with the parameter attached to it" produces the resulting value.

That concept doesn't exist in most languages. For example, in Haskell,

```
g = (+) 3
```

applies the function (+) to the argument 3, and binds the result to "g". The function (+) takes a number parameter A, and returns a new function that takes a different number parameter B and returns A+B. Thus "g" is bound to a function that takes a number B and returns B+3.

In this view, functions can return functions, and functions take functions as parameters. So there is no such thing as "binding a parameter to a function" here, just calling a function with a parameter, which produces a result.

To get a better understanding of the differences in view, it's useful to look at a simple implementation of a lambda expression evaluator in Haskell, where you'll see that the primitive pieces of code are functions (lambdas), constants, and function calls – there isn't any such thing as a "function with a parameter attached". Then compare to LISP evaluators where quoted lists are literally considered to be "functions with parameters attached", which can later be evaluated to produce results.

## 2.50 [slashdot] Already there

If you're running a 3D-accelerated PC game or modelling application, the majority of your computer's FLOPS are already consumed by a non Von Neumann computing device.

For better or worse, most of the PlayStation2's computing power is locked up in a non Von Neumann architecture.

So the evolution of computing to non Von Neuman architectures isn't so much news as a gradual shift that began about 5 years ago with 3dfx, and is really starting to happen large-scale right now.

The justification for FPGAs in consumer computing devices could be seen as a generalization of the rationale behind 3D accelerators: they bring you the ability to get a 10X-100X speedup in certain key pieces of code that are inherently very parallel and have very predictable memory access patterns.

I think the timeframe for mainstream FPGA style devices is quite far off, though. They need to evolve a lot before they'll be able to beat the combination of a Von Neumann CPU augmented with several usage-specific non Von Neumann coprocessors (the GPU, hardware TCP/IP acceleration, hardware sound...)

Here are the major issues:

- You'll need a lot more local memory than these devices have now – there is a very limited set of useful stuff you can compute given a 32K buffer (a la PS2) and significant setup overhead.

- The big lesson from CPU's (and I expect from GPU's in the next few years) is that things REALLY flourish once you have virtualization of all resources, with a cache hierarchy extending from registers to L1 to L2 to DRAM to hard disk. For virtualization to make sense with FPGAs, Star Bridge's quoted reprogram times (40 msec) would need to improve by about 10,000X. Without this, you can really only run one task at a time, and that task can only have a fixed number of modules that use the FPGA.

Even then, it's not clear whether the FPGAs will be able to compete with massively parallel CPU's. In 3 more process generations, you should be able to put 8 Pentium 4 class CPU's on a chip, each running at over 10 GHz, at the same cost as current .13 micron CPU. Such a system would be VERY easy to program, a couple orders of magnitude more so than an FPGA. So even though it wouldn't have as much theoretical computing power as an FPGA, massively parallel CPU's are likely to win out because they have the best cost/performance when you factor in development cost.

## 2.51 [comp.lang.functional] Re: curried function calling

In any ordinary programming situation, functions of the form  $a \rightarrow b \rightarrow c$  are isomorphic to functions of the form  $(a,b) \rightarrow c$ . In other words, you can trivially translate functions and calls from one form to the other and back without losing any information or changing the semantics.

So the only practical question is, which form is more convenient for a given situation?

- Map tends to be more convenient in curried form because often you're looking for a function  $[a] \rightarrow [b]$ , and can conveniently obtain the right one with `map(f)`. Whereas, in uncurried form, you would need to express this as `lambda(x::[a]) -> map(f,x)` which is more verbose.

- For functions like addition, you almost always call it with two both arguments present, so `add(a,b)` may be more convenient than `add(a)(b)`. Some languages bring additional benefits to uncurried form. For example, with support for variable-length arrays, you could just as easily define "add" to take an array of numbers and have `add(a,b,c,d)=a+b+c+d`, `add(a,b)=a`, `add(a)=a`, and `add()=0` which are often convenient.

*I suppose the type signature of # would be  $\# : (() \rightarrow A) \rightarrow A$  except it's really a special operator, and not a function according to this system.*

In LISP, that kind of operator is useful because you can first-class write code (in your language itself, not just in the compiler) that traverses these "unevaluated expressions" and transforms them in interesting ways. This is so powerful in LISP because it's done in an untyped way: you can traverse an expression and turn it into anything imaginable, and if there are any type violations, they only show up at runtime. Thus compile-time typechecking doesn't impose any limitations on how you can traverse stuff.

In a strongly-typed and simply-typed language like Haskell, your # operator can be implemented directly in the language as `"call :: lambda(f::t-`

> u)-> lambda(x::t)-> f(x)”, which obeys the rule “call(f)(x)=f(x)” and therefore “call(f)=f” – so it turns out to just be the identity function. So, it doesn’t add any expressive power to the language. The type system restricts you to only writing code that’s well-typed, so the neat expression-traversal features of LISP go away.

Overall, I think the conclusion is that the “#” operator brings your language new capabilities if and only if your language is untyped.

*This is ugly, I know. I only ask because I’m interested in language features and I’m just thinking about different ways to construct functions, and this idea came up.*

LISP from the 1970’s pretty much represents the apex of untyped languages. It does all of the interesting things that are doable and does them very simply, leaving most of the further debate to subjective issues like syntax.

The really interesting and hard problems outstanding in language design are in extending the power of strongly-typed languages. Here are some cool papers I recommend checking out, covering language features that do add significant new expressive power:

<http://www.cs.caltech.edu/~jyh/papers/fool3/default.html> <http://ttic.uchicago.edu/~dmcalister/ontic-spec.ps>

## 2.52 [fa.haskell] Implementing RefMonads in Haskell without ST,IO

Given the common definition of RefMonad:

```
class Monad m => RefMonad m r | m -> r where
  newRef :: a -> m (r a)
  readRef :: r a -> m a
  writeRef :: r a -> a -> m ()
```

2.52. [FA.HASKELL] IMPLEMENTING REFMONADS IN HASKELL  
WITHOUT ST,IO

Is it possible to actually implement a working instance of RefMonad in Haskell, without making use of a built-in monad like IO or ST? If so, I'd love any tips – I've been making good use of monads for a while, but can't figure this one out.

The Java programmer in me wants to implement RefMonad by passing around a function from integers (think of them as pointers or heap indices) to "objects", and in readRef, "cast" the "object" to the appropriate type "t".

If it's not possible to implement a typesafe RefMonad instance directly in Haskell, without making use of built-in imperative features like IO, then doesn't this refute the claims that monads implement imperative features functionally?

-Tim

## 2.53 [fa.haskell] Re: Implementing RefMonads in Haskell without ST,IO

Hi Derek,

How can one implement RefMonad below directly in Haskell?

```
class Monad m => RefMonad m r | m -> r where
  newRef :: a -> m (r a)
  readRef :: r a -> m a
  writeRef :: r a -> a -> m ()
```

I've been able to implement a monad that encapsulates "references to integers", by creating a monad that passes "the heap" (a function `int->int`, from heap indices to integer values stored in the heap) in and out of functions. That was pretty straightforward, and the monad makes everything look like an imperative language that supports updatable references to integers (and only integers).

2.53. [FA.HASKELL] RE: IMPLEMENTING REFMONADS IN HASKELL WITHOUT ST,IO

But how can one implement `RefMonad` to support references of all possible types simultaneously?

The problem is that `readRef` needs to return elements of an arbitrary type, so it has to be able to extract them from some sort of "heap" function or data structure. But what does the heap data structure look like? The values that `readRef :: r a -> m a` depend on type "a", and I can't figure out how to implement that, because "a" is local to that declaration.

If this were implemented in C++, for example, the heap could just store "objects", and `readRef` could cast the object to the appropriate type when returning it. But in Haskell, I don't see any way to do this.

-Tim

## 2.54 [fa.haskell] class MonadPlus m => MonadRing m

```
class MonadPlus m => MonadRing m where
  mone :: m a
  ** :: m a -> m a -> m a
```

Does anybody have experience with a construct like `MonadRing` above? I'm interested in prior implementations, examples of instances, and coherence rules.

A conceptual example: consider `ListMonad`, used to implement "non-determinism", or computations that may produce multiple values. In this context, the order or duplication of elements in a list is not significant; we think of them as sets of values:

From `Functor`:

```
fmap f xs -- apply a function to all elements in the set, and collect the results in
```

### 2.54. [FA.HASKELL] CLASS MONADPLUS M => MONADRING M

From Monad:

```
xs>>=f -- apply a set-returning function to all elements of x, and merge all the results
return xs -- generate a singleton set containing only x
```

From MonadPlus:

```
mzero -- the empty set
xs++ys -- the union of sets xs and ys
```

From MonadRing:

```
xs ** ys -- the intersection of the sets xs and ys.
mone -- the unit of **. conceptually, "the set of all sets". With
```

Haskell's type system, this notion isn't feasible with ListMonad directly, but could be used with similar "set-like" monads containing elements drawn from a finite set.

For set-like monads, there is then a further notion of a partial order relation defined by set inclusion. This could be defined along the lines of "mcontains :: m a -> m b -> m bool". For ListMonad, mcontains xs ys can be defined as true iff xs contains all elements of ys, in order, allowing arbitrary new elements that happen to occur between them.

I'm not sure of the practical utility of these constructs in the Haskell world, but I've been working with them for a while in an experimental language and have found them very powerful, especially when coupled with first-class types with set-like functionality a la David McAllester "Ontic" language. In my particular case, I have found the following syntactic sugar to aid a great deal in the readability of complex monad expressions:

```
-- Non-monad constructs:

f(x) -- call function f with parameter x
let x=v in e -- let-binding.

-- Functor syntactic sugar:

f($x) -- like map f x
```

2.54. [FA.HASKELL] CLASS MONADPLUS M => MONADRING M

```
let x$=v in e -- like map (\x -> e) x

-- Monad syntactic sugar:

f(%x) -- like x >>= f
let x%=v in e -- like v >>= \x -> e
```

This could be considered a syntactic alternative to the "do" notation and list comprehension notation.

-Tim Sweeney

## 2.55 [comp.lang.functional] Re: Is Strictness a Hack?

In the presence of `_|_`, all types are inhabited and all values are type-theoretically equal. In its absence, proofs-as-propositions works, and lazy evaluation is observationally equivalent to strict evaluation. So, clearly, `_|_` is the point of controversy, and not lazy vs strict.

It would be good for future languages that aspire to be pure to allow a clear distinction between expressions that admit `_|_` and expressions that disallow it. The former can be used in propositional proofs, proof-carrying-code, and other metamathematical formulations regardless of evaluation strategy, while the later are necessary for many mainstream programming tasks.

Effects are not monads and `_|_` is not an effect. We should view these concepts as if we have a languages with a base axiom system that doesn't admit `_|_` or effects. And then we have additional axioms that we can add in to the evaluation of terms in particular contexts, to capture the additional evaluation semantics.

Adding axioms can sometimes make the type theoretic equality relationship more specific (for example, in allowing the distinction of terms which

differ only in effects not visible under the base axioms), and sometimes narrow it, for example by collapsing it into a single term  $\_|\_$ .

## 2.56 [comp.lang.functional] Re: Is Strictness a Hack?

*I \*think\* I understand and agree with you when you say that the point of controversy is  $\_|\_$  rather than lazy vs strict, but I am unclear why "all values are type-theoretically equal" in the presence of  $\_|\_$ . Can you please elaborate?*

This isn't something you run into when writing and running programs in languages like Haskell. Rather, it comes up when trying to prove properties about them in a manner independent of the evaluation rules. If a language admits  $\_|\_$ , then you can write a function that polymorphically "returns" an element of any type (here, I mean it claims to return such a thing, but actually never returns). For example, if unbounded recursion is allowed, then you can write:

```
f(t:type):t = f(t)
```

Given that, you can "produce" an element of an uninhabited type. Specifically, if  $t$  is the type of propositions that  $a=b$ , then  $f(t)$  is a proof that  $a=b$ , for any possible  $a$  and  $b$ . So, when a type system allows  $\_|\_$ , you can prove anything.

*It would be good for future languages that aspire to be pure to allow a clear distinction between expressions that admit  $\_|\_$  and expressions that disallow it. The former can be used in propositional proofs, proof-carrying-code, and other metamathematical formulations regardless of evaluation strategy, while the latter are necessary for many mainstream programming tasks.*

*Do you mean to exchange "former" and "latter" in the second sentence above? Assuming so, I believe Daniel Wang and I both agree with you.*

Oops, yes.

*I also believe Daniel Wang and I both agree with the rest of your post, except there seems to be a difference in terminology definition when you say that*

*Effects are not monads and `_|_` is not an effect.*

*It is quite clear to me that `_|_` is a monad (namely the monad taking each type  $x$  to  $x + \text{unit}$ ). The term "effect" is rather vague, and while I am happy to take monads to be a technical approximation to the intuitive notion, I am also happy to entertain alternative definitions. Can you explain yours a bit more?*

Well, you could model the notion of "a term that is either `_|_` or a pure value" using Haskell's Maybe monad. This is similar to the notion of `_|_` that comes up when considering functions that throw exceptions (without associated values) that can be caught upstream in a program, as some parser combinators do.

But, `_|_` comes up in other situations where Maybe doesn't capture the essence of what's happening. Consider the type of (potentially infinite recursive) functions from integers to integers, for example. If you model these as `f :: Int -> Maybe Int`, then you would expect to be able to write code like

```
a = f 7
b = case a of
    Nothing -> ..
    Maybe x -> ..
```

Here, we've tried to hide `_|_` (the low-level potential for no-return) in the Maybe monad. We would expect "f" to be a pure function that is certain to return in finite time, and to either return a value or a Nothing tag. How do you generate such an "f" from a traditional recursive function definition? That would require solving the halting problem.

Of course, that's not possible, so any language that tries to hide `_|_` in a monad like `Maybe` is either impossible to implement, or would have to be incapable of expressing general recursion.

BUT:

If you were to modify the Haskell compiler, you could still use such a monad to encapsulate terms that potentially evaluate to `_|_`. You'd probably just do this with a newtype aliasing the identity monad, and only allowing calling of generally recursive functions using that monad.

This situation would be similar to the `RefMonad` parts of `IO` and `ST`. There is some consensus (still unproven) that it's not possible to actually implement typesafe references directly with Haskell monads. But monads do provide a clean way of exposing built-in runtime features like this and limiting the scope where they can be used.

## 2.57 [fa.haskell] Re: Typesafe MRef with a regular monad

Conjecture: It's impossible to implement `RefMonad` directly in Haskell without making use of built-in `ST` or `IO` functionality and without `unsafe` or potentially diverging code (such as `unsafeCoerce`).

Any takers?

If this is true or suspected to be true, any thoughts on whether a structure besides `Monad` could encapsulate safe references in Haskell without core language changes? My intuition is that no such structure exists in Haskell with power and flexibility equivalent to `RefMonad` (support for references of arbitrary types not limited by their context.)

If this is generally thought to be impossible in Haskell, what sort of language extensions would be needed to make this work safely, meaning without unsafe coercions? This seems like a hairy problem. Yet it gets to the core question of whether Haskell's monads can really implement imperative features (such as references) in a purely functional way, or are

just a means of sequentializing those imperative constructs that are built into the runtime.

Any solutions I can think of require a dependent-typed heap structure, and that all references be parameterized by the heap in which they exist.

-Tim

## 2.58 [fa.haskell] Re: Safe and sound STRef [Was Implementing RefMonads in Haskell without ST,IO]

Oleg,

This is a very neat solution to providing coercion-free references in a local environment.

I'm trying to generalize this to some sort of Monad-like typeclass, where there is a nice mapping from Monad's to this new and more powerful typeclass, so that one can combine typed references, IO, etc., into a single framework.

It seems to me that your approach couldn't be generalized in this way in Haskell, because the type of the resulting reference-using "computation" depends on the precise set of heap operations performed there. So, for example, you couldn't do something like:

```
..  
a <- newRef Int 123  
b <- if (some conditional) then (newRef Int) else (a)  
..
```

Because the heap types propagated out of the conditional's two branches differ.

The only way I can see generalizing your technique to support this sort of thing requires a type system supporting both dependent types and subtyping. Think the reference monad as looking somewhat like the state monad; instead of a single piece of state, it propagates a dependent-typed pair of (heapTypeFunc,heapValueFunc) similar in spirit to your PList construct, with the type guaranteeing that any heap operation returns an output heapTypeFunc that's a contravariant extension of its input heapTypeFunc.

Conjecture: Implementing type-safe (coercion-free), composable computations over typed references isn't possible in Haskell. By composable I mean that some operator similar in spirit to  $> > =$  on Monads can be implemented and that computations with differing effects can occur in if-branches.

But then again, I had not thought the problem you solved using PList to be solveable in Haskell, and am very eager to be proven wrong!

-Tim

## 2.59 [comp.lang.functional] Re: Is Strictness a Hack?

Neel: Google for "RefMonad", "newIORef", "readIORef", etc. The notion is that RefMonad encapsulates typesafe references to mutable values stored on a heap, passed around in Monadic style using IO or ST, which are instances of a new typeclass RefMonad denoting "monads capable of storing references".

*To make the discussion clearer I think it would be better to use some name other than 'Maybe' for this monad, perhaps 'NT' for non-termination.*

*Haven't you tried to write a function  $b$  of type*

*$NT Int \rightarrow Int$*

*or something similar? Surely this is no more possible than writing a function of type*

*IO Int -> Int*

*The point of wrapping it up in a monad is that you cannot then try to pattern-match or otherwise 'escape' from the nontermination. Yes, Maybe is a monad, but it is a data type first and the monad machinery is just a syntactically convenient way of handling it.*

*(There could be an unsafePerformComputation function of type NT x -> x as an analogue unsafePerformIO. You'd use it when you were certain that a computation really would terminate, despite its being in the NT monad.)*

This works, but to do this meaningfully, you'd need to modify the Haskell compiler to assure that it's impossible to write potentially infinite-recursive functions unless you're computing within an NT monad, right?

The problem is, that rules out the use of general fixpoint recursion, so you'd have to expose some other kinds of constructs for more limited recursion schemes, such as primitive recursion. This makes it VERY hard to write programs. For example, this definition is no longer allowable:

```
fac :: integer -> integer
fac 0 = 1
fac n = n * fac (n - 1)
```

Such a framework might be useful for proofs-as-programs research, but I can't imagine writing significant software without general recursion.

## 2.60 [slashdot] A few notes

Regarding documentation, check out the Unreal Developer Network for a huge amount of documentation.

Also, the 3D Buzz team has created many excellent training videos covering many aspects of the Unreal tech, from programming to content creation.

*Ultimately, the developer of such a mod should be fairly compensated based on the popularity and ultimately, the sales, of their mod, not a one-time payout.*

The mod developer keeps complete ownership of his work. The contest doesn't take that away.

For example, if you enter an early version of your mod in the contest, you could later create a retail game based on it and pursue a publishing deal. The Tactical Ops mod for the original Unreal Tournament went this route and was published in retail by Atari.

Regarding tax issues, one should definitely consult a tax attorney upon making the finals for the grand prize. My understanding (IANAL) is that, if we gave you a \$350K cash prize, that would be revenue for your mod team's corporation or small business. If you then spent that \$350K on an Unreal engine license with the intent of using it commercially (which is the only reason one would want such a license), you would then incur a \$350K expense, leaving a net tax liability of zero. So a direct award of an engine license is not necessarily a taxable event.

## **2.61 [comp.lang.functional] Re: local variables don't cause side effects**

Marshall,

This is actually a great question. Pay no attention to the functional programming curmudgeons (referential transparency isn't a religion, it's a tool).

From the outside, a function that has no side-effects or dependence on external state or I/O is referentially transparent and can be treated uniformly regardless of whether its internal implementation is purely functional or locally imperative.

From the inside – when you get to evaluating the function's body – whether the code is purely functional or (at least partly) imperative has a huge

2.61. [COMPL.LANG.FUNCTIONAL] RE: LOCAL VARIABLES DON'T  
CAUSE SIDE EFFECTS

impact on your evaluation strategy. For example, lazy evaluation can be a reasonable strategy when evaluation functional code, whereas evaluation order must be precise and explicit in imperative code.

In a language that supports both paradigms, being able to distinguish "functional functions" and "imperative functions" is very valuable. For one example, in an implicitly parallelizing compiler, a call to a purely functional function can be handed off to another thread; if many such calls exist in a region of code with no intermingled side-effects, then N calls can be handed off to N independent CPU's without worrying about their global evaluation order – even if those functions happen to use locally imperative code internally (for example, by manipulating a local heap that's discarded upon return).

I've been able to mix and match both paradigms in an experimental compiler and am very happy with the results. My feeling is that for real-world development projects carried out by mainstream programmers (i.e. games, graphical user interfaces, large-scale server apps), imperative features are essential to a language's success.

But allowing both functional and imperative approaches to be mixed, in the local-vs-global approach you suggest, gives you the best of both worlds.

## 2.62 [comp.lang.functional] Re: local variables don't cause side effects

*If you can construct references to local variables, and allow assignments via references, a function can indeed modify variables outside its scope, so you have to be rather careful about what you allow programmers to do with locals. But let me assume that all mechanisms that might modify a nonlocal variable were disabled, to keep the answer interesting ;-)*

For those interested, there is a great body of research on this topic, the general idea being that multiple heaps can exist; pointers carry around "which heap do I point to" information, and one is prevented (at the

typechecking level) from writing to someone else's heap. Google for "Typed Regions".

*But you don't need them. To transform an imperatively-written function into a functional version: 1) Whenever the imperative code reassigns to a local variable, just create another variable that will take up the new value. 2) Whenever there's a loop, move the loop body into a recursive helper function. (Note most loops that you'll encounter can be transformed into higher-order functions which will do the recursion for you. You don't have to reinvent the wheel in every case.)*

This works for the case of loops within a single function which "update" loop variables during iteration. Such constructs can always be translated into tail-recursive functions quite easily.

But that's not the case, in general, with imperative code. For example, if one desires the ability to dynamically allocate local objects within a function and manipulate those objects as well as mutable references to them, in that function, and in other nested functions, with the general ability to update such references through closures, then the translation to purely functional code is far more complex. Haskell has solved the problem in purely functional exposure of a behind-the-scenes native implementation quite elegantly (google "Haskell IO Monad" and "RefMonad"). But the case of a purely functional typesafe verifiably divergence-free mechanism is an open research problem that probably requires a more advanced type system to solve.

Overall, I think the notion of "translating arbitrary imperative programs into purely functional ones" is a very valuable way of conceptualizing, theorizing, and reasoning about things like mutable references and exceptions. But it shouldn't be considered an implementation technique, because it's vastly more complex and less efficient than just exposing imperative functionality in the language itself.

## 2.63 [comp.lang.functional] Re: local variables don't cause side effects

*For one example, in an implicitly parallelizing compiler, a call to a purely functional function can be handed off to another thread; if many such calls exist in a region of code with no intermingled side-effects, then N calls can be handed off to N independent CPU's*

...

*I find this possibility fascinating.*

*Don't hold your breath: home users have little if any use for that. Unless they are heavily into gaming, in which case the CPU load is most likely split between mainboard and graphics card, which is a scenario that's quite the opposite of parallelizing function calls.*

Lots of bulk data processing operations besides low-level rendering are highly parallelizable. Examples from game development: video codecs, audio codecs (a significant CPU hit when you're playing 20 simultaneous MP3/OGG-format ambient sound and special effects streams), physics solvers, collision detection algorithms, scene traversal algorithms. Many of these algorithms are scalable to the extent that if we had significantly more CPU power, we could make games that are significantly more interesting and impressive.

*That's just a side note. The \*real\* problem is that handing off computations to a different processor often entails more work than just stacking the call. Compilers may try to predict which calls should be parallelized and which shouldn't, but mispredictions in that area will quickly eat up the advantages gained here.*

Agreed. On current popular architectures, the overhead of thread communication is such that you can only benefit from handing off 10,000+ CPU cycle operations to threads. In this case, some form of explicit parallelism is certainly needed because it's surprisingly hard to write a code

generator that can guess, from looking at a piece of code, how much time it is likely to take, even within several orders of magnitude. Whereas, a reasonably experienced programmer is far better at judging such things.

That practical parallelism has to be explicit for the time being doesn't negate the merits of this approach though. In C++, to split up a complex algorithm into multiple threads, I not only have to explicitly create those threads, I have to write and debug a complex algorithm that is free of deadlocks and sharing violations, which is terribly difficult and error-prone in the presence of the kind of nondeterminism present in typical multithreaded C++ code.

On the other hand, consider a language where one can specify that a particular function is globally side-effect free and have the typechecker verify that it really is. Then when you annotate the text program with parallelism hints, the compiler can verify that those hints can be safely applied. Thus that a program is deadlock-free can be mechanically verified in the same way that a typechecker verifies that a program is type-mismatch-free.

*Besides, you usually don't have  $N$  processors, you have at most two. Maybe four, in the next five years or so - but I wouldn't bet on that. Besides, even with SMP, if two processors share the same memory bus, memory latencies tend to eat up another gob of the performance advantage.*

*In principle, it's all a great idea. In practice, there are so many little hurdles that take away a percent here, a percent there that it's not so great anymore.*

In 1999, this was unquestionably true. In 2003, this is basically true as far as practical consumer software deployment goes. By 2005, it will be mostly false thanks to symmetric multithreading, and by 2007 it will be rendered a complete lie :-)) by multicore CPU's with very large caches.

*But allowing both functional and imperative approaches to be mixed, in the local-vs-global approach you suggest, gives you the best of both worlds.*

## 2.63. [COMPL.LANG.FUNCTIONAL] RE: LOCAL VARIABLES DON'T CAUSE SIDE EFFECTS

*The one thing that I've been missing for two years now is that nobody has yet shown a convincing example of this: How to write code that a) is (possibly heavily) imperative b) has a purely functional interface c) allows the compiler to infer (b) even in the presence of (a).*

*For example, I'd like to use imperative data structures to build functional containers. No way, even though I'm prepared to annotate all the loops with loop invariants to help the compiler along...*

I've solved the much more modest problem of checking side-effect-free function declarations which do imperative things internally, by using something along the lines of "region types" for local mutable objects, and requiring some explicit source annotations of pointer regions and scopes of effects. This approach works and is relatively easy because whatever imperative stuff happens inside your function invocation is guaranteed to have vanished when your function returns, i.e. by disallowing closure-capture and so on.

Regarding example here, that's a great example and a much harder problem, which might serve as a torture test of a language claiming to fully mix functional and imperative programming safely.

Perhaps if you could prove (in the Curry-Howard sense) that your module is observationally equivalent (in the Leibniz equality sense) to a purely functional module type satisfying the same identities, then the compiler could accept it as being an instance of that purely functional module. A bunch of issues immediately come to mind:

- From an operational point of view, this approach has got to be regarded as a coercion that introduces things such as synchronization primitives into the module's function. Just going through and proving that each function in the module satisfies its invariants invokes a bunch of hidden assumptions, such as that the code is executed in a single-threaded manner.

- But proving that it's deadlock-free, deadlocks being another manifestation of  $\_|\_$ , would seem to impose additional restrictions on what is allowed in the module. If each invocation of a functional-datatype-with-

imperative-implementation requires single-threaded invocation, then any sort of polymorphic behaviour in such datatypes could lead to deadlocks that are impossible to detect at compile time.

Overall, this sounds promising, but whether or not it becomes practical will depend on the severity of the restrictions, because functional-modules-implemented-imperatively wouldn't likely be accepted if it introduced undetectable deadlocks in code.

## 2.64 [comp.lang.functional] Re: local variables don't cause side effects

*If each invocation of a functional-datatype-with-imperative-implementation requires single-threaded invocation, then any sort of polymorphic behaviour in such datatypes could lead to deadlocks that are impossible to detect at compile time.  
I'm not sure how this could happen. Could you give an example*

Say you've implemented functional-style arrays using imperative techniques (heaps, mutability, sequential execution), and have somehow proven to the compiler that your implementation is observationally equivalent to a functional implementation. What if a given function (for example, array concatenation) contains a loop that leaves some linked lists temporarily inconsistent, even though it's guaranteed to be fix everything up by the time it exits. Wouldn't the code generator have to then insert locks at entry/exit from the function to assure atomicity? What if your imperative implementation had to call some user-defined function passed into it as a parameter, and it called that function at a point when it left a linked list in an inconsistent state? What if that user-defined function happens to access the same data structure that is currently in an inconsistent state?

The real question is: what limitations would a type system have to impose on programs in order for it to admit purely-functional-constructs-with-opaque-imperative-implementations without introducing potential

deadlocks or nondeterminism? (An open question, I think.)

## 2.65 [slashdot] This is lame

Given that Blizzard is using the trademarks "StarCraft" and "WarCraft" in this very specific market (realtime computer strategy games), their claim that "FreeCraft" infringes on their copyright is reasonable and very likely winnable in court.

So, no problem, just rename FreeCraft to a unique name that clearly isn't derivative of Blizzard's product. And don't be mad that they asked you to do this, because they have the right and obligation to protect their copyrights.

On the other hand, unless you've physically ripped code or content out of StarCraft or WarCraft and put it in your game, any claim that your game is "too similar" to theirs seems absurd and almost certainly has no basis in copyright or trademark law. If you ignore them on that issue, then they are almost certain to go away.

And if they don't go away nicely, the resulting outrage over their persecution of the open source community would almost certainly force them to go away ashamedly.

But if you just cave in, and you fail to stand up for your rights when presented with this sort of threat, then you are certain to lose your rights.

If a person asks you to get out of his seat, you move. If a bully asks you to give up YOUR seat, you fight.

## 2.66 [comp.lang.functional] Terminology: sgn, abs, normal, magnitude

On real numbers, the "abs" and "sgn" are commonly defined as:

$$\begin{aligned} \text{abs}(x) &= x \text{ if } x > 0, 0 \text{ if } x = 0, -1 \text{ if } x < 0 \\ \text{sgn}(x) &= 1 \text{ if } x > 0, 0 \text{ if } x = 0, -1 \text{ if } x < 0 \end{aligned}$$

In many kinds of vector space (in this case, a 3d vector space), the "magnitude" and "normalize" operations are commonly defined as:

$$\begin{aligned} \text{magnitude}(v) &= \sqrt{v.x^2 + v.y^2 + v.z^2} \\ \text{normalize}(v) &= v / (\sqrt{v.x^2 + v.y^2 + v.z^2}) \text{ if } v \neq 0, 0 \text{ otherwise} \end{aligned}$$

These operations obey the identities:

$$\begin{aligned} \text{abs}(x) &= x * \text{sgn}(x) \\ \text{sgn}(x) &= \lim_{y \rightarrow x} y / \text{abs}(y) \end{aligned}$$

$$\begin{aligned} \text{magnitude}(v) &= v * \text{normalize}(v) \text{ (where } * \text{ is the inner product operation)} \\ \text{normalize}(v) &= \lim_{w \rightarrow v} w / \text{magnitude}(w) \end{aligned}$$

So, from a certain point of view these two sets of operations represent the same notion: "absolute value" and "magnitude" represent the directionless size of a number or vector, while "sign" and "normalize" represent the sizeless direction of a number or vector.

QUESTION: Is there a name for or formalization of this generalized notion of "size" and "direction"? And exactly what is the minimal set of spaces upon which these operations are generally defined?

From my informal Google search on this topic, it sounds like the "normed vector spaces" (<http://mathworld.wolfram.com/NormedSpace.html>) are what I'm looking for. If this is the answer, then would it be considered kosher to view the real numbers as a normed vector space (over the real numbers themselves), where the norm is simply the absolute value?

The context of the question is this: I'm writing a computer math library, and am looking for the appropriate set of types over which the `sgn/abs/magnitude/normalize` functions can be defined generally, and am looking for an abstraction and nomenclature which will offend the fewest people possible.

## 2.67 [TYPES] Type theory vs floating-point arithmetic

I'm looking for pointers to attempts to reconcile the differences between type theories and the data types and functions defined by the IEEE 754 floating-point standard.

For example, IEEE 754 defines equality over floating point numbers in a way that conflicts with Leibniz equality. There exist two IEEE "zeros",  $-0$  and  $+0$ , and the standard requires that they compare as if they are as equal, though there exist functions that distinguish between them. Additionally, IEEE defines a class of "not-a-number" values  $x$  such that  $x=x$  is deemed false.

This would seem to require that a language either expose two equality operators (a natively-supplied Leibniz equality operator, and a "IEEE 754 numerical equality" operator which differs at some values), or that the language disobey the floating-point standard with regard to equality. Java takes the later route, while C/C++ and the Haskell report appears to be silent on the issue.

There is a further problem regarding subtyping. It might first appear that the type of 32-bit floating point values is a subtype of the type of 64-bit floating point values. Though one can losslessly convert from 32-bit to 64-bit and back, performing a 32-bit arithmetic operation on such 32-bit numbers can produce a different result than performing a 64-bit arithmetic operation on their 64-bit extensions, and then converting back. So from a type theoretic point of view, the types of 32-bit and 64-bit floating point numbers must be disjoint.

Finally, IEEE 754 mandates that a compliant computer architecture and language pair expose a mechanism for reading and writing a set of "sticky

flags” which indicate whether any imprecise, overflowed, or invalid operations have been encountered since the flags were last reset; and a set of ”rounding modes” affecting subsequent operations. These concepts seem incompatible with referential transparency and lazy evaluation.

## 2.68 [sci.math] Computational algebraic integers

Two motivating examples:

- One can construct a finitary computational model of the integers in many ways. One way is to represent them as a string of binary digits, with the classic two’s-complement algorithms for binary addition, subtraction, multiplication, division, and comparison. If the representation is constrained to have no redundant leading digits, then the mapping from integers to representations is bijective.
- Given a computational model of the integers, one can construct a finitary model of the rational numbers, by representing each rational number as a pair (numerator,denominator). The elementary operations can be defined in the obvious way, i.e.  $(a,b)+(c,d)=(a*d+b*c,b*d)$ . If the representation is constrained to be in lowest terms with positive denominator, the mapping from rational numbers to representations is bijective.

Now a few questions:

- Does there exist a finitary computational model of the algebraic numbers, such that there exists a bijection between mathematical algebraic numbers and their model, and there exist terminating algorithms for addition, subtraction, multiplication, division, and comparison? To clarify ”finitary” here, I mean a representation such that if you start with a finite set of rational numbers and perform a finite sequence of field operations and polynomial-solving operations on them, then the result is guaranteed to be representable in finite space.
- Does there exist a halting algorithm that, for every finite set of polynomial coefficients  $(a_0+a_1*x+a_2*x^2+..=0)$ , can determine all of its solutions in the algebraic numbers in such a representation?

- Any references to computer implementations of the above?

Thanks for any pointers.

## 2.69 [comp.lang.functional] Re: result of heterogeneous union

*What should be the result type of a union of the two tables?  
Specifically, what is the type of the resulting set?*

- *None; this causes a type error*
- *(int|string)*
- *Most specific supertype of int and string. (Possibly 'alpha'.)*
- *alpha, the maximal supertype.*

This depends on your type system.

In a type system like that of C++, this sort of typing scenario isn't expressible.

In a C# style language with boxing, you could best say `union(t,u)=the most specific type that is a supertype of both t and u`. Thus `union(int,string)=object`.

To translate that most simply to Java, you would need to avoid basic types like `int`, and only take unions of class type. So you might say `union(Int,String)=Object`, though it's not quite possible to express this in Java directly, even with the long-awaited generic type extensions.

In the functional programming world, if you start with a simply-typed language without subtyping, then add unions (and the resulting subtype relations) in the obvious way, `union(string,int)` would be a supertype of both `int` and `string` but of no other types besides subtypes of `int` and `string`. This seems the cleanest approach which doesn't rely on higher-order typing constructs. Here, you would most naturally have `union(set(t),set(u))=set(union(t,u))`.

However, it's possible to be even more precise. In a type system like McAllester's Ontic language (see <http://ttic.uchicago.edu/~dmcallester/>)

[ontic-spec.ps](#)), you could say that, for any value  $a$ , the type of  $a$  is just "the-set-of-all  $a$ ", and never have ambiguity in questions of "what is the type of this expression in this context?" This approach is far outside of existing programming language practice, but I recommend reading the paper, just to see what's possible at one far extreme of the design space.

## 2.70 [sci.math.symbolic] Re: Repeated exponentiation $n@m$

Also, google for "Knuth's arrow" operator. It's the general case of addition/multiplication/exponentiation for arbitrary iterations. Also related is Ackermann's function.

As far as I understand it, these operations and their inverses are closed over the complex numbers, so no new number systems need to be introduced to study them completely.

## 2.71 [comp.lang.functional] Re: result of heterogeneous union

*However, it's possible to be even more precise. In a type system like McAllester's Ontic language (see <http://ttic.uchicago.edu/~dmcallester/ontic-spec.ps>), you could say that, for any value  $a$ , the type of  $a$  is just "the-set-of-all  $a$ ", and never have ambiguity in questions of "what is the type of this expression in this context?" This approach is far outside of existing programming language practice, but I recommend reading the paper, just to see what's possible at one far extreme of the design space.*

*I'm \*very\* interested to read this paper, so I downloaded it, and also searched for more references. I didn't find too much; the first hit on google was for a submission to the excellent Lambda*

*the Ultimate weblog by one Tim Sweeney. Sadly it did not spur much discussion.*

Someday I think it will be recognized as one of the most important ideas in programming language foundations, aside Church's lambda calculus and the Curry-Howard isomorphism. Admittedly nobody else including the author seems to share this view.

*Unfortunately, when I try to convert this file into anything I can view or print (pdf, say) the fonts turn to mush. I could probably read it anyway if I was 15 years younger, but not with the eyes I have now.*

*I know it's totally off-topic, but mightn't anyone have any idea how to convert that .ps file into something with readable fonts? (Alternatively, if you know how I can become 15 years younger, that would also get a big "thank you.")*

For reading and printing .ps files, try GhostScript and GSView, from: <http://www.cs.wisc.edu/~ghost/>

*Does this idea of "nondeterminism" have anything to do with constraint-based programming languages, such as Mozart?*

Yes, in fact I think the Ontic style (though not necessarily syntax) provides a far more expressive framework for constraint-based languages than existing practice.

The important thing in understanding Ontic, and perhaps the reason it hasn't become widely-known, is that its use of the word non-determinism doesn't imply an unpredictable outcome, but merely that a term may have multiple values (in the same order-independent and unique-up-to-equality way that a set may contain multiple values), and that sharing is handled properly in the presence of multivaluedness. In other words, a term like  $\{x:=3|4,y:=x+1\}$  is equivalent to  $\{3,4\}|\{4,5\}$  and does not include the value  $\{4,4\}$ .

2.71. [COMPLANG.FUNCTIONAL] RE: RESULT OF HETEROGENEOUS UNION

## 2.72 [comp.lang.functional] Re: Python from Wise Guy's Viewpoint

*THE GOOD:*

*THE BAD:*

1. *f(x,y,z) sucks. f x y z would be much easier to type (see Haskell)*  
90% of the code is function applications. Why not make it convenient?

9. *Syntax for arrays is also bad [a (b c d) e f] would be better than [a, b(c,d), e, f]*

Agreed with your analysis, except for these two items.

#1 is a matter of opinion, but in general:

-  $f(x,y)$  is the standard set by mathematical notation and all the mainstream programming language families, and is library neutral: calling a curried function is  $f(x)(y)$ , while calling an uncurried function is  $f(x,y)$ .

- "f x y" is unique to the Haskell and LISP families of languages, and implies that most library functions are curried. Otherwise you have a weird asymmetry between curried calls "f x y" and uncurried calls which translate back to "f(x,y)". Widespread use of currying can lead to weird error messages when calling functions of many parameters: a missing third parameter in a call like  $f(x,y)$  is easy to report, while with curried notation, "f x y" is still valid, yet results in a type other than what you were expecting, moving the error up the AST to a less useful obvious.

I think #9 is inconsistent with #1.

In general, I'm wary of notations like "f x" that use whitespace as an operator (see <http://www.research.att.com/~bs/whitespace98.pdf>).

## 2.73 [comp.lang.python] Re: Python from Wise Guy's Viewpoint

*THE BAD:*

2.72. [COMPLANG.FUNCTIONAL] RE: PYTHON FROM WISE GUY'S VIEWPOINT

1. *f(x,y,z) sucks. f x y z would be much easier to type (see Haskell) 90% of the code is function applications. Why not make it convenient?*
9. *Syntax for arrays is also bad [a (b c d) e f] would be better than [a, b(c,d), e, f]*

Agreed with your analysis, except for these two items.

#1 is a matter of opinion, but in general:

-  $f(x,y)$  is the standard set by mathematical notation and all the mainstream programming language families, and is library neutral: calling a curried function is  $f(x)(y)$ , while calling an uncurried function is  $f(x,y)$ .

- " $f x y$ " is unique to the Haskell and LISP families of languages, and implies that most library functions are curried. Otherwise you have a weird asymmetry between curried calls " $f x y$ " and uncurried calls which translate back to " $f(x,y)$ ". Widespread use of currying can lead to weird error messages when calling functions of many parameters: a missing third parameter in a call like  $f(x,y)$  is easy to report, while with curried notation, " $f x y$ " is still valid, yet results in a type other than what you were expecting, moving the error up the AST to a less useful obvious.

I think #9 is inconsistent with #1.

In general, I'm wary of notations like " $f x$ " that use whitespace as an operator (see <http://www.research.att.com/~bs/whitespace98.pdf>).

## 2.74 [comp.lang.functional] Re: Object Identity

*\*Copying\* may not be meaningful, but identity is.*

*(2) Identity breaks observational equivalence.*

*Not true. It depends on the equivalence predicate.*

By comparing functions "by pointer" (or, more generally, by their internal representation), you can sometimes determine that two functions *\*are\**

observationally equivalent. But you can't generally determine that two functions *aren't* observationally equivalent. Two functions might have different internal representations, while still representing the same function, from an extensional equivalence point of view.

In theory, this matters whenever a language's equality predicate corresponds to observational equivalence, a.k.a. Leibniz equality.

In practice, this matters wherever a language runtime might perform runtime code specialization (optimizing some occurrences of a function but not others, thus changing some internal representations), might serialize the function for sending across the network, etc.

One should be very careful with any piece of code that claims to compare two functions for equality. If in a Turing-complete language such an equality predicate exists on functions, then there are cases where it's going to either throw an exception or return false when in theory it should return true. Such an equality can be a useful engineering tool, but it's not mathematical equality.

## 2.75 [sci.math] Re: Ordering a Power Set

*This question arises from economics, but I'll give you the question first, the context later: Suppose we have a set of  $n$  objects, and a binary relation  $S$  that is a linear ordering, ie irreflexive, asymmetric, complete, transitive. Is there any way to use this relation to induce some sort of ordering on the power set of our set of objects?*

Yes: Associate each (finite) set with a sorted array containing exactly those elements in that set, and use the lexicographic (dictionary) order of those arrays. It will also be a linear order with a minimum element (the empty array) but no maximum element.

## 2.76 [fa.haskell] Re: lifting functions to tuples?

In case this is of interest, there is another novel way that a language might support liftTup in full generality. Example code:

```
> LiftTup(f:function)(a:f.dom,b:f.dom)={f(a),f(b)}
> f(x:int)=x+1
> LiftTup(f)(3,7)

{4,8}

> LiftTup(caps)("Hello","Goodbye")

{"HELLO","GOODBYE"}
```

Here, the type system supports full set-theory-style subtyping, with functions subtyping such that  $a \rightarrow b < : c \rightarrow d$  iff  $b < : d$  and  $c < : a$ ; an empty type "false" containing no elements, and the "function" type the type of functions with empty domain and thus a supertype of every function type. Thus "LiftTup" above takes a parameter  $f$  which may be a function of any domain and range.

The " $f.dom$ " notation extracts the (dependent) domain type of the function  $f$ , which depends on the actual function being passed. Thus the code above is statically typecheckable, and the program, i.e. `LiftTup(caps)(3,7)` would fail, because "caps" is a function from strings (arrays/tuples of characters) to strings.

Though, in my language, tuples are merely dependent-typed arrays of known size, which themselves are a subtype of the type of dependent-typed arrays of unknown (i.e. existentially quantified) size, which are expressed syntactically as, i.e. `[]:int`. So the above can be rewritten more generally:

```
> Map(f:function)(a[]:dom.f)=array(i:nat<a.len)f(a)
```

```
> Map(f) (3,7,9,12)
```

```
{4,8,10,13}
```

Where the "array" notation is an array lambda expression, and `a.len` extracts the length of an unknown-sized array.

This language allows quite a bit more generality and type precision in function and data type definitions, though code tends to lack the conciseness made possible by Haskell type deduction.

-Tim

## 2.77 [sci.math] Universal set theory and three-valued logic

One apparent way of avoiding the paradoxes of naive set theory is to turn set-defining characteristic functions into partial functions from sets to the three-valued logic  $\{T,F,\text{bot}\}$ . This three-valued logic extends classical logic in the obvious way with  $\text{and}(T,\text{bot})=\text{or}(F,\text{bot})=\text{bot}$ ,  $\text{and}(F,x)=F$ ,  $\text{or}(T,x)=T$ ,  $\text{not}(\text{bot})=\text{bot}$ , so that every truth function has a fixed point.

Obviously the law of excluded middle does not hold:  $\text{or}(a,\text{not}(a))$  only implies that  $a$  is  $T$  or  $\text{bot}$ . This gives the logic a constructive character.

In my formulation, I identify each set with its characteristic function from sets to  $\{T,F,\text{bot}\}$ . Thus given  $s:\text{set}$ , the membership of an element  $x$  can be tested with  $s(x)$ . In the general case, this is a partial function, returning  $\text{bot}$  for some values. I call such sets "partial sets", and sets whose characteristic function is in  $\{T,F\}$  "total sets". Every set of ZF and NF is a total set, with this theory admitting a strictly larger class of sets than either.

Russell's set  $R=\{s:\text{set}|\text{not}(s(s))\}$  is then a partial set. All ZF sets are elements of  $R$ , while some elements of NF are not elements of  $R$ , while some new sets such as  $R$  itself are of undecidable membership.

I've translated the ZF axioms to this set theory, rephrasing them in terms

of characteristic functions and new for-all and there-exists logic operators performing logical conjunction and disjunctions across all elements of a characteristic functions. Everything appears to be sound and avoids known paradoxes.

With the new axioms, it is easy to construct a bijection from the universal set to its power set. Cantor's proof that  $|P(x)| > |x|$  for all non-empty sets  $x$  proceeds by constructing  $C = \{a : x \mid \text{not}(P(x)(a))\}$  and using the law of excluded middle to derive a contradiction on its membership in  $P(x)$ . This goes away for lack of excluded middle, leaving  $C$  a partial set which appears not to be constructively contradictory.

The one worrying aspect of this approach is that it identifies sets with characteristic functions from sets to logic values:  $\text{Set} = \text{Set} \rightarrow \{T, F, \text{bot}\}$ . I have only been able to develop an intuition of such sets in a purely constructive way, by writing down a finite list of possibly self-referential equations defining sets, and convincing myself that a unique solution exists. This is much in the style of NF's axiom that every (possibly cyclic) graph corresponds to a set, but I allow unlimited comprehension.

Are there any known problems with this approach to set theory? Any pointers to research on the topic?

Tim Sweeney

## 2.78 [sci.math] Re: Universal set theory and three-valued logic

*With the new axioms, it is easy to construct a bijection from the universal set to its power set.*

*How?*

In a set theory with a universal set  $U$  and sufficient power to reason about it,  $P(U) = U$ . So, the identity function is a suitable bijection.

If  $P(U) = U$  is controversial (given a set theory with a universal set and no urelements), I would be interested in hearing so. I realize that some uni-

versal set theories preclude analysis of  $P(U)=U$  on the grounds of stratification, but if a theory allows one to reason about  $P(U)=U$  then surely it must either be true or undecidable.

Note that Cantor's construction showing that for non-empty  $x$ ,  $|P(x)| > |x|$ , fails on such "large sets" as  $U$ : the set one constructs to refute the possibility of a bijection involves negation and the law of excluded middle, so the set membership relation of the refutation set may be partial, and that partiality precludes the contradiction.

## 2.79 [sci.math] Re: Universal set theory and three-valued logic

*In my formulation, I identify each set with its characteristic function from sets to  $\{T, F, \text{bot}\}$ . Thus given  $s$ :set, the membership of an element  $x$  can be tested with  $s(x)$ . In the general case, this is a partial function, returning bot for some values. I call such sets "partial sets", and sets whose characteristic function is in  $\{T, F\}$  "total sets". Every set of ZF and NF is a total set, with this theory admitting a strictly larger class of sets than either.*

*In some ways, this is the structure of a topos, or at least a functor from a topos category to something similar. Is the approach meant to be categorial (which, by the way, is a good thing in my eyes – I'm just curious)? Then, your partial classification appears to mainly distinguish the topos of sets from some of the many other topos.*

My goal is to show evidence that a particular type theory is reasonable, by equating it to some existing system. An axiomatic set theory was my first hope due to the relative simplicity, but a topos will do.

*The categorial study of paradox is becoming a large field these days, and it appears you may be repeating some of the work already done (which can be soooo frustrating sometimes!). I don't*

*mean to assume any level of study, but perhaps I might suggest that, if you haven't, you should check out some of the resources available in this field. There are articles available online I can suggest.*

Do you have any pointers to research on topoi coinciding to a set theory with a universal set, or more generally to any topos-centric analysis of paradoxes due to nonwellfoundedness?

*I could make one suggestion, it would be to not restrict yourself to your trivalent logic. Any Heyting algebra is possible, and expands your research into the much more fruitful world that all topoi present. In fact, because of natural distinctions that present themselves between propositions that take on the "middle" value, trivalent theories are often looked at only as summarisations of a more natural infinitely valent theory. Good resources for this can be found in intuitionist discussions, but it is more general.*

In my setting, I am mainly interested in mechanically analyzing sets to determine whether they are inhabited and, if so, whether they are singletons. A trivalent logic seems sufficient for this: any finer structure can be projected down for my purposes.

*Also, may I ask why you posted to `comp.lang.functional`? This intrigues me because some of my own research has been around the evaluation of the lambda calculus and proof / evaluation theory in the context of non standard logics, but I do not see this approach explicitly stated in your message.*

I'm developing a programming language with enough expressive power that one may easily express the major paradoxes from set theory, both those due to recursion (the liar paradox) and those due to nonwellfoundedness or reflection (Russell's paradox). I have a compiler that analyzes programs and assigns to each term a set cardinality approximations classifying its potential inhabitation, as well as an indicator of the decidability of this classification.

The general idea is that single-valued terms correspond to runtime-executable parts of programs, while more general (potentially uninhabited or multivalued terms) may be bundled up into extensional types and reasoned about within the limits of the compiler's resolution rules. This all works to the extent of being able to express the paradoxes, recognize their undecidability, and mechanically determine that, for example, certain concrete terms do or don't belong to Russell's set. But though it tends to work well in mechanical terms, it lacks a theoretical basis.

Thank you very much for the pointers!

Tim Sweeney

## **2.80 [comp.lang.functional] Re: Perfect list shuffling: an interesting algorithm**

Can't one always do shuffling with  $O(n)$  element accesses, by forming a new array one element at a time, at each point selecting and removing (by index) a random element from the original array? For example, one could implement this in C by starting with an array of  $n$  elements, selecting a random item from among the remaining contiguous items, appending that item onto the result array, and then replacing it (in place) with the item at the end of the array, repeating until all elements are exhausted.

This appears to me to require  $O(n)$  element accesses, and a source of  $O(n \log n)$  random bits. Any source of  $O(n \log n)$  computational steps – unless you are treating each random bit access as a step – would I think be an artefact of using functional lists instead of indexable, updatable arrays.

Tim Sweeney

## 2.81 [comp.lang.functional] Re: Computing alpha-equality in pure lambda calculus

Klaus,

Structural equality of unevaluated lambda terms is solveable in the pure lambda calculus. One easy way to implement this is to store terms using de Bruijn indices (Google for it) and to compare their representations for structural equality. If comparing for equivalence is a big goal of a system you're building, check out Luca Cardelli's "Explicit Substitutions" paper for a nice term evaluation and representation scheme that does not rely on names.

## 2.82 [TYPES] Functors and free theorems

A covariant functor is an operator  $F: \text{type} \rightarrow \text{type}$  paired with a function  $\text{Map}: (t \rightarrow u) \rightarrow (F(t) \rightarrow F(u))$  such that  $\text{Map}(\text{id}(t)) = \text{id}(F(t))$  and  $\text{Map}(f \cdot g) = \text{Map}(f) \cdot \text{Map}(g)$  where  $\cdot$  is function composition and  $\text{id}(t)$  is the identity function over type  $t$ .

Given a functor's  $\text{Map}$  function, since  $\text{Map}(\text{id}(t)) = \text{id}(F(t))$ , we can at least in concept recover a functor's operator  $F$  from its  $\text{Map}$  function by translating the resulting identity arrow  $\text{id}(F(t))$  back to its unique type. So a  $\text{Map}$  function obeying the two laws above is sufficient to define a functor.

Question:

Are there type systems where the type of all such a  $\text{Map}$  functions (for all possible functors) can be expressed precisely?

I realize that in a type system supporting proofs-as-programs and equality types, it is possible to express the types of the proofs of  $\text{Map}(\text{id}(t)) = \text{id}(F(t))$  and  $\text{Map}(f \cdot g) = \text{Map}(f) \cdot \text{Map}(g)$  directly. But this takes the form of a product of a function and two relatively verbose proofs about it. Is there a way to do this more directly?

I'm looking for something similar in spirit to the guarantee that in the Hindley-Milner type system, every function inhabiting the type  $\text{ForAll}(t:\text{type}).t \rightarrow t$  is an identity function. Is there a way to express the type of Map functions such that a parametricity-derived free theorem guarantees that it is inhabited exactly by those functions that are correct Map functions of functors? A solution in any type theory whatsoever (not just Hindley-Milner) would be of interest.

Tim Sweeney

## 2.83 [comp.lang.functional] Re: My take on Monads

Monads are a purely functional technique for doing what you just described, but also a lot more. In Haskell, the language on which much of the development of monads has been focused, the IO monad is the monad that implements this "pass the state of the world in and out of every function" pattern.

Many other interesting programming patterns are possible with monads. For example, Haskell's State monad enables you to write self-contained blocks of code using exceptions and heap allocations whose access is guaranteed to be constrained within that block of code, so that a function implemented this way is purely functional "on the outside", though it uses many imperative-style techniques "on the inside".

Also check out Haskell's Array monad (for computations producing multiple values) and Maybe monad (for computations that may return an optional failure code) to get a feeling for some very different programming techniques encapsulated by monads.

## 2.84 [comp.lang.functional] Combining lazy and eager evaluation of terms

In an attempt to combine some of the benefits of lazy and eager evaluation, I have implemented a language with an evaluation strategy which is strict with respect to divergence, but performs lazy evaluation on certain intermediate subterms to allow a more expressive use of recursion.

To be more precise:

- By "strict with respect to divergence", I mean that for every possible function or data constructor  $f$ ,  $f(\text{bot})=\text{bot}$ . Thus for lists,  $\text{Cons}(\text{bot},x)=\text{Cons}(x,\text{bot})=\text{bot}$ .
- Performing "lazy evaluation on intermediate subterms" enables one to successfully express recursive data structures like  $x=\text{Cons}(3,x)$  for an infinite list, or  $r:=\text{NewRef}(\text{Cons}(3,r))$  for a circular linked list.
- Unlike lazy languages supporting strictness declarations, and strict languages supporting laziness declarations, this language has no such declarations and always follows a single evaluation scheme.

Currently, my implementation constructs thunks for all potentially recursive terms, traversing them in applicative-order yet deferring access to the value of any reentered thunk (a thunk whose evaluation is already in progress) until its value is strongly required according to normal-order evaluation rules. Thus every subterm is always fully evaluated eventually, but certain recursive terms are supported which would, in an eager language, either be disallowed or result in access to uninitialized data.

The benefits of this approach are are:

- Recursion and self-reference can be used much more expressively than in traditional eager languages.
- The far majority of "traditional" eager functional and imperative code can be automatically deemed safe for unboxed eager evaluation by the compiler at performance comparable with a mainstream language. In particular, any code block which exhibits only backward references, or for which a topological sort can be automatically derived. Note that any code for which this property doesn't hold could, in a language like C++ or

Java, access uninitialized variables.

- The complex process of class/module loading and runtime linking in languages like Java and C# can be implemented as plain old execution of the modules' top-level code, for example with proper resolution of circular module references that construct constants. Recent papers on Java class loaders in purely eager environments indicate that this process is somewhat contrived in the absence of general support for thunked evaluation.
- The scheme can be combined with an effects-tagging and "applicative-order readiness-tagging" strategy to support the free intermixing of functional and imperative constructs, with effects-free terms potentially evaluating in arbitrary order (due to recursion or compiler optimizations) while guaranteeing that imperative effects only execute sequentially, with any recursive sequentiality violations sometimes being statically detectable, but always being dynamically detectable. Such an imperative framework can be implemented either in a monadic Haskell-style IO/fixIO "top-level" framework, or in a more traditional imperative language.

The drawbacks are:

- To a programmer reading a piece of code, it is not always obvious whether that code will invoke thunked evaluation, yet this distinction may have a significant impact on the resulting performance.
- The evaluation scheme is more complex than either a traditional normal-order or applicative-order evaluator, in particular in its treatment of transitions between thunked lazy evaluation and unboxed eager evaluation.
- Every intermediate term (except for the untaken branch a conditional) is fully evaluated. This leads to more work than a pure lazy evaluator, which discards intermediate terms which don't contribute to the final result. I do not consider this drawback significant to a mainstream programming audience, however it rules out the use of some techniques popular among lazy functional programmers, such as infinite non-circular lists and streams.

The interesting problems related to this approach are:

- The conservative, compile-time detection of subterms where a tradi-

tional eager runtime evaluation strategy may be safely invoked to maximize performance. This is a more lenient problem than traditional strictness analysis, because there are no functions for which  $f(\text{bot}) \neq \text{bot}$ . Thus the question is not "Can this term be evaluated to ground here?" but "Can this term be evaluated without reaching any unevaluated circular dependencies?" Note that the recursion operator is the only construct that can prevent a program from being evaluated with the optimized eager scheme, but that some uses of recursion are detectably safe under an unboxed strict evaluation scheme, such as " $x = \text{Cons}(3, \text{Cons}(\text{Head}(x), \text{Nil}))$ ".

- The runtime unboxing of thunked values at all transition points where a subterm requiring internal thunk-based evaluation moves into a context allowing eager evaluation. For example, in " $x = \text{Cons}(3, x), y = \text{Head}(x)$ ", we can evaluate  $x$  using thunks, unbox the results, and evaluate  $y$  traditionally. This unboxing process becomes tricky with function closures; in the general case, two versions of a closure's environment and code pointer need to be kept around to support its evaluation in both both thunked and unboxed contexts, and in some cases only the thunked version may be safely invoked. There are various optimizations possible here – such as dynamically generating a closure's thunked environment when only an eager environment is available – but the tradeoffs are quite complex.

Does anybody have any references to work on similar evaluation schemes?

Tim Sweeney

## 2.85 [TYPES] Combining lazy and eager evaluation of terms

In an attempt to combine some of the benefits of lazy and eager evaluation, I have implemented a language with an evaluation strategy which is strict with respect to divergence, but performs lazy evaluation on certain intermediate subterms to allow a more expressive use of recursion.

To be more precise:

- By "strict with respect to divergence", I mean that for every possible

function or data constructor  $f$ ,  $f(\text{bot})=\text{bot}$ . Thus for lists,  $\text{Cons}(\text{bot},x)=\text{Cons}(x,\text{bot})=\text{bot}$ .

- Performing "lazy evaluation on intermediate subterms" enables one to successfully express recursive data structures like  $x=\text{Cons}(3,x)$  for an infinite list, or  $r:=\text{NewRef}(\text{Cons}(3,r))$  for a circular linked list.

- Unlike lazy languages supporting strictness declarations, and strict languages supporting laziness declarations, this language has no such declarations and always follows a single evaluation scheme.

Currently, my implementation constructs thunks for all potentially recursive terms, traversing them in applicative-order yet deferring access to the value of any reentered thunk (a thunk whose evaluation is already in progress) until its value is strongly required according to normal-order evaluation rules. Thus every subterm is always fully evaluated eventually, but certain recursive terms are supported which would, in an eager language, either be disallowed or result in access to uninitialized data.

The benefits of this approach are are:

- Recursion and self-reference can be used much more expressively than in traditional eager languages.

- The far majority of "traditional" eager functional and imperative code can be automatically deemed safe for unboxed eager evaluation by the compiler at performance comparable with a mainstream language. In particular, any code block which exhibits only backward references, or for which a topological sort can be automatically derived. Note that any code for which this property doesn't hold could, in a language like C++ or Java, access uninitialized variables.

- The complex process of class/module loading and runtime linking in languages like Java and C# can be implemented as plain old execution of the modules' top-level code, for example with proper resolution of circular module references that construct constants. Recent papers on Java class loaders in purely eager environments indicate that this process is somewhat contrived in the absence of general support for thunked evaluation.

- The scheme can be combined with an effects-tagging and "applicative-order readiness-tagging" strategy to support the free intermixing of func-

tional and imperative constructs, with effects-free terms potentially evaluating in arbitrary order (due to recursion or compiler optimizations) while guaranteeing that imperative effects only execute sequentially, with any recursive sequentiality violations sometimes being statically detectable, but always being dynamically detectable. Such an imperative framework can be implemented either in a monadic Haskell-style IO/fixIO "top-level" framework, or in a more traditional imperative language.

The drawbacks are:

- To a programmer reading a piece of code, it is not always obvious whether that code will invoke thunked evaluation, yet this distinction may have a significant impact on the resulting performance.
- The evaluation scheme is more complex than either a traditional normal-order or applicative-order evaluator, in particular in its treatment of transitions between thunked lazy evaluation and unboxed eager evaluation.
- Every intermediate term (except for the untaken branch a conditional) is fully evaluated. This leads to more work than a pure lazy evaluator, which discards intermediate terms which don't contribute to the final result. I do not consider this drawback significant to a mainstream programming audience, however it rules out the use of some techniques popular among lazy functional programmers, such as infinite non-circular lists and streams.

The interesting problems related to this approach are:

- The conservative, compile-time detection of subterms where a traditional eager runtime evaluation strategy may be safely invoked to maximize performance. This is a more lenient problem than traditional strictness analysis, because there are no functions for which  $f(\text{bot}) \neq \text{bot}$ . Thus the question is not "Can this term be evaluated to ground here?" but "Can this term be evaluated without reaching any unevaluated circular dependencies?" Note that the recursion operator is the only construct that can prevent a program from being evaluated with the optimized eager scheme, but that some uses of recursion are detectably safe under an unboxed strict evaluation scheme, such as `"x=Cons(3,Cons(Head(x),Nil))"`.
- The runtime unboxing of thunked values at all transition points where a

subterm requiring internal thunk-based evaluation moves into a context allowing eager evaluation. For example, in "x=Cons(3,x),y=Head(x)", we can evaluate x using thunks, unbox the results, and evaluate y traditionally. This unboxing process becomes tricky with function closures; in the general case, two versions of a closure's environment and code pointer need to be kept around to support its evaluation in both both thunked and unboxed contexts, and in some cases only the thunked version may be safely invoked. There are various optimizations possible here – such as dynamically generating a closure's thunked environment when only an eager environment is available – but the tradeoffs are quite complex.

Does anybody have any references to work on similar evaluation schemes?

Tim Sweeney

## 2.86 [comp.lang.functional] Re: Combining lazy and eager evaluation of terms

*What about from n = Cons n (from (n+1))*

This function is nonterminating (when called) in this evaluation scheme. Every term is reduced to (an immutable reference to) a ground value. This is done in such a way that circular terms like "x = Cons 3 x" are admitted, but unending computations are not. For example, even "from n = Cons n (from n)" is nonterminating (when called), because "from n" is a function call that initiates a new reduction.

However, I haven't rigorously investigated questions about the interaction between beta/eta reduction and recursion in a system like this.

## 2.87 [comp.lang.functional] Re: Combining lazy and eager evaluation of terms

*Do you want to say that replicate x = let xs = Cons x xs in xs is terminating but not equivalent to replicate x = Cons x (replicate*

2.86. [COMPLANG.FUNCTIONAL] RE: COMBINING LAZY AND EAGER EVALUATION OF TERMS

$x$ )?

Yes, though this looks disturbing as written. The distinction is clearer if we write the recursive binders explicitly using the lambda/mu notation. We can say this is terminating:

```
lambda(x).mu(xs).Cons(x,xs)
```

but this is non-terminating:

```
mu(f).lambda(x).Cons(x,f(x))
```

If I understand all of the ramifications correctly, one could say that, in this evaluation scheme, "Data structures may safely be self-referential, but all calling sequences must be finite." I would love to hear any feedback on the soundness of this approach.

## 2.88 [TYPES] Combining lazy and eager evaluation of terms

Thanks to everyone who responded on and off the list to this query. Here is a summary of direct responses on combining lazy and eager evaluation of terms:

Derek Dreyer pointed to his paper on supporting many useful forms of recursion in a call-by-value language, framed in the context of ML:

*"A Type System for Well-Founded Recursion"* <http://www-2.cs.cmu.edu/~dreyer/papers/recursion/popl.pdf>

*In this paper I use a pure call-by-value evaluation, but I design the type system so that it ensures that when evaluating a recursive definition (like `val rec x = e`), that `x` will not be \*dereferenced\* when evaluating `e`, although it may be \*referenced\*. Dereferencing means you actually want to get at the underlying value (i.e. the value that `e` ends up evaluating to), whereas referencing simply means you want a "pointer" to that (as-yet-undefined) value. (I say "pointer" because if the underlying*

*value is already a boxed object, there's no need to rebox it in the underlying implementation.) This is useful, for instance, for purposes of separate compilation, where each definition in a pair of mutually recursive definitions can be evaluated separately given a pointer to the other one. However, no high-level language that I'm aware of actually makes a distinction between the notions of "referencing" and "dereferencing" of recursive variables.*

Paul Hudak pointed out some undesirable properties of the evaluation strategy I proposed:

*A desirable property of most languages is that they have some kind of a "least fixed-point" semantics. But yours seems to lack that. In particular, you say:*

*By "strict with respect to divergence", I mean that for every possible function or data constructor  $f$ ,  $f(\text{bot})=\text{bot}$ . Thus for lists,  $\text{Cons}(\text{bot},x)=\text{Cons}(x,\text{bot})=\text{bot}$ . But Performing "lazy evaluation on intermediate subterms" enables one to successfully express recursive data structures like  $x=\text{Cons}(3,x)$  for an infinite list, or  $r:=\text{NewRef}(\text{Cons}(3,r))$  for a circular linked list.*

*But the least fixed-point of the equation  $x = \text{Cons}(3,x)$  is bottom, according to the first bullet, i.e. not the infinite list of 3's.*

Henrik Pilegaard pointed out the Hope language described at <http://www.soi.city.ac.uk/~ross/Hope/>. Hope performs eager evaluation of functions, but supports lazy data structures.

*Here are some other references that I found very useful:*

*"Implementation of Non-Strict Functional Languages" Kenneth R. Traub [http://www.amazon.com/exec/obidos/tg/detail/-/0262700425/qid=1094235827/sr=1-1/ref=sr\\_1\\_1/104-9051043-1871907?v=glance&s=books](http://www.amazon.com/exec/obidos/tg/detail/-/0262700425/qid=1094235827/sr=1-1/ref=sr_1_1/104-9051043-1871907?v=glance&s=books)*

*This 1991 book covers a broad variety of topics on practical implementation, and describes a model known as "lenient evaluation", a non-strict evaluation model that makes efficiency*

*gains by not supporting full lazyness. In this model, parameters are be passed in to functions unevaluated when necessary to make progress on evaluation. This model can support cyclic data structures like "x:=Cons(x)" and dependent data structures like "x:=Pair(2,x.first+1)", but not acyclic infinite data structures such as "letrec f(n:int):=Cons(n,f(n+1)) in f(0)".*

*"How much Non-strictness do Lenient Programs Require?" Klaus Schauer, Seth Goldstein <http://www.cs.ucsb.edu/schauser/papers/95-fpca.ps>*

*This describes the lenient evaluation scheme of the Id90 functional language, designed for efficient execution of code on parallel architectures, and analyzes the expressive power of various forms of non-strictness.*

*Guy Tremblay has a number of interesting papers on lenient evaluation at <http://www.info.uqam.ca/~tremblay/>.*

Some final thoughts on the topic:

Lenient evaluation appears to be an attractive model for future mainstream programming languages, because it enables much of the expressiveness of lazy evaluation (such as self-referential data structures) without running into many of the fundamental limitations of strictness analysis that prevent lazy languages from performing competitively with strict languages; and because it enables a more natural implementation of parallel evaluation than either lazy or strict languages.

However, while implementations of lenient evaluation are widespread (in Id90, pH, Octopus, and others), I've been unable to find an analysis of the semantics of lenient evaluation. This seems an important outstanding problem because, as Paul Hudak points out, there are real problems with a direct application of domain theory to such languages because all functions have the property  $f(\text{bottom}) = \text{bottom}$ . Any pointers to papers on this topic would be quite welcome.

Another worrying property of a lenient evaluation scheme is that, though infinite computations are prohibited, one can still construct cyclic values like  $x = \text{Cons}(3, \text{Cons}(4, x))$ . In a traditional eager language, it is easy to prove that a function like Map (on lists) is convergent. Here, convergence depends on the list being finitary. This also implies that any built-in

equality operation must either perform bisimulation (rather than simple recursive comparison) or also be potentially divergent. Has there been any investigation into this topic? I could imagine inserting tests, upon each reduction of a thunk to head-normal form, to assure that any non-finitary data structures cause divergence at creation-time, therefore assuring that functions like Map can count on only receiving acyclic inputs. But this seems quite inefficient.

Tim Sweeney

## 2.89 [TYPES] Parametricity with subtyping and a Top type

Does subtyping and the existence of a Top type necessarily break parametricity?

Parametricity guarantees that every function of type  $\forall a. a \rightarrow a \rightarrow \text{Bool}$  is a constant function. The same derivation appears applicable to  $\forall a. a \rightarrow a \rightarrow \text{Top}$  also, implying that it too must be constant. But  $\lambda x. \lambda y. x$  and  $\lambda x. \lambda y. y$  are distinct functions inhabiting  $\forall a. a \rightarrow a \rightarrow \text{Top}$ .

What has gone wrong here?

Pierce's thesis touches on Top in F/\ but doesn't seem to address parametricity.

Does there exist a systematic study of type system extensions known to be compatible or incompatible with parametricity?

Tim Sweeney

## 2.90 [TYPES] Subtyping, extensional equality, and contravariance

This is a query about a new direction in type systems supporting subtyping, extensional equality, and contravariance.

Consider a type system with types "int" (the integers) and "nat" (the natural numbers) with  $\text{nat} < : \text{int}$ . Now define:

```
abs :: int -> int
abs x = if x >= 0 then x else -x

id :: int -> int
id x = x
```

Traditionally, one would say that  $\text{abs} \neq \text{id}$  (at type  $\text{int} \rightarrow \text{int}$ ), and  $\text{abs} = \text{id}$  (at type  $\text{nat} \rightarrow \text{nat}$ ). In other words, extensional equality is only meaningful "at a type  $t$ ", and the extensional equality of values may differ depending on the types at which they are interpreted.

But I am interested in a type system in which equality represents the same relation at all types, in other words:  $a = b$  (at type  $t$ ) implies that  $a = b$  (at type  $u$ ) for all types  $u$  containing  $a$  and  $b$ . Thus we can talk of  $a = b$  without referring to an interpretation type  $t$ . The effect of this is to make all values non-polymorphic, that is, their interpretation is independent of the context in which they are interpreted.

To satisfy this goal, I found it necessary to associate with every function value a unique "domain type" which is extractable and is carried around with the function value:

```
dom :: forall t,u. t -> u -> type
dom f = t
```

Then, two functions  $f$  and  $g$  are equal iff  $\text{dom}(f)=\text{dom}(g)$  and for all  $x:\text{dom}(f)$ ,  $f(x)=g(x)$ . In the examples above,  $\text{dom}(\text{abs})=\text{dom}(\text{id})=\text{int}$ , and  $\text{abs}\neq\text{id}$ .

The "invariant function space" type (written  $t \rightarrow u$  for now) then means the type of functions  $f$  in which  $\text{dom}(f)=t$  and for all  $x:t$ ,  $f(x):u$ . Thus we have  $\text{abs}:\text{int} \rightarrow \text{int}$ , but not  $\text{abs}:\text{nat} \rightarrow \text{nat}$ , because  $\text{dom}(\text{abs})=\text{int}$ .

The "contravariant function space" type  $t \multimap u$  then means (intuitively) the union, for all supertypes  $t'$  of  $t$  and all types  $u'$ , of all functions  $f:t' \rightarrow u'$  in which for all  $x:t$ ,  $f(x)$  belongs to type  $u'$ . In other words, all functions whose domain includes  $t$ , and whose image under  $t$  belongs to  $u$ . This is interesting because we then have  $\text{abs}:\text{int} \multimap \text{int}$  and  $\text{abs}:\text{nat} \multimap \text{nat}$ , but not  $\text{abs}:\text{int} \multimap \text{nat}$ .

Given an empty type "empty" and any type  $t$ , the type "empty  $\rightarrow t$ " is then the type of all functions; name the later "function".

This approach appears to enable implicit parametric polymorphism without the existing of any (implicit or explicit "forall") quantifiers. For example, in Haskell one would write:

```
map :: (t -> u) -> [t] -> [u]
map f xs = [f x | x <- xs]
as = map abs [3,-5,7]
```

If we took the dependent-typed language Cayenne and removed parametric polymorphism, to write this we would have to specify all types explicitly:

```
map (t::type) (u::type) (f::t->u) (xs::[t])
as = map int int abs [3,-5,7]
```

But with the type system above, where every function value carries its exact domain type along with it, we can write the following:

## 2.90. [TYPES] SUBTYPING, EXTENSIONAL EQUALITY, AND CONTRAVARIANCE

```
map (f::function) (xs::[dom(f)]) = [f x | x <- xs]
as = map abs [3,-5,7]
```

Note that the above "map abs" application is as concise as in Haskell, but requires no forall quantifiers (neither explicit nor implicit), as the function's domain type is extracted from the function's value.

We can also define a universal type "top" containing all possible values. And then a Haskell function like:

```
id :: t -> t
id x = x
```

Can be translated without quantifiers as:

```
id (x::top) = x
```

Has this approach been taken before? I would appreciate any references to work along these lines, especially results on parametricity.

Tim Sweeney

## 2.91 [TYPES] semantics of 'quote'

I looked into this a while ago and found two ways of approaching a "quote" operation.

The first is to allow one to dynamically convert any value to its Godel number or something similar (e.g. an AST representation). Such an operation has a sweeping impact on the language's properties.

For example, adding "quote" to a language with extensional equality would make its equality inconsistent with observable equivalence: `quote(f)` and `quote(g)` would reveal that `f=lambda(x:int)=x+1` and `g=lambda(x:int)=1+x` are distinct.

Adding "quote" to a language respecting the parametricity property would break parametricity: You would write a function like `forall(t:type) lambda(x:t) quote(x)` which converts a value of any type to an integer and thus distinguish values of universal type.

The second, and less devastating, approach would be to expose "quote" as a purely syntactic feature, where you could write code like:

```
f=quote {lambda(x:int) x+1 }
```

And then, from `f`, you could extract both the semantic value and the syntactic value. For example, `f.value` would return the function `lambda(x:int) x+1`, and `f.quote` would return a Gödel representation, textual representation, abstract syntax tree representation, or some other representation that does not obey the parametricity and extensionality properties you would expect the value itself to obey.

The extent of the "quote" operation here is just the syntactic span of code inside the quote. For example, in `forall(t:type) lambda(x:t) quote {x}`, the quote `{x}` would just return some syntactic indicator of a reference to a bound variable (a local property), rather than introspecting the actual value of `x` and returning some syntactic representation of it (a global property violating extensionality and parametricity).

This issue is important to many languages, because programmers have a general desire for both "nice properties" (parametricity, extensionality, etc) and for the ability to extract metadata from their program in order to enable automation of certain operations (graphical user interface creation, implementing persistence, etc). An explicit syntactic quote operation like this enables both capabilities to be mixed and matched explicitly, without violating nice global properties of programs.

-Tim