Subject: Re: REDUCE
Posted by Kenneth Mankoff on Fri, 30 Mar 2001 01:52:30 GMT
View Forum Message <> Reply to Message

> The question, to all you C-programmers: is there a better way?
[snip]
> ...the code logic to compute the maximum will be the same, both
> symbolically for all types for many types, in the compiled code itself.

Hi JD,

hmmm... not 100% sure, but wouldn't c++ templates solve this problem?

And for the cases where it is "symbolically" the same but not "compiled
the same", I'm not sure what this means, but I'm guessing you would handle
these cases with overloading your operators.

Of course, C isn't C++, so this might not help.

I can provide code examples and more info if you wish.

-k.

 ---------- ---------
   Kenneth Mankoff   LASP
mankoff@lasp.colorado.edu   1234 Innovation Drive
     303.492.3264   Boulder, CO 80303
 ---------- ---------

On Thu, 29 Mar 2001, JD Smith wrote:

>
> IDL Monkeys:
>
> If you've done any C+IDL programming, skip to the end for a (possibly
> impossible) programming challenge: WARNING: not for the faint of heart.
>
> The little n-dimensional histogram thought exercise revved my juices to
> waste more time and finish a little project which had been languishing
> for months: REDUCE. For those of you without idelible memories of all
> things idl-pvwave, I was lamenting the lack of any good, multipurpose,
> built-in threading tool, i.e. something for applying operations over a
> given dimension of a multi-dimensional array, similar to the way "total"
> allows you to specify a total'ing dimension. Craig has also supplied us
> with cmapply, which, while useful, is forced to compromise speed.
>
> The result is a C-program for building as a DLM and linking with IDL.
> Currently, the options supported are:

>
> Operations:
> "MAX"
> "MEAN"
> "MEDIAN"
> "MIN"
> "MULTIPLY"
> "TOTAL"
>
> Options:
> "DOUBLE" - work in double precision
> "EVEN"   - for median, same as keyword in IDL's median() function
>
> That is, you can take the median over the third dimension of a 5D
> hypercube, and so on.
>
> REDUCE works in any native numeric type, preserving type if possible.
> For certain operations, namely, TOTAL, MEAN, and MULTIPLY, it is always
> performed in FLOAT (or DOUBLE if passed or natively present in the
> input). This is to avoid overflow, and follows the example of IDL's
> total(). I may add a keyword "NATIVE" to force working in the native
> type, overflows be damned, which might be useful in some instances.
>
> Things I like about REDUCE:
>
> 1. It doesn't screw with your type unless it has to. For instance,
> IDL's median() converts everything but byte to float first. Why?
> REDUCE respects your right to use ULON64's or what have you natively.
>
> 2. It's fast. Preliminary testing indicates its from 2-50 times as
> fast as the same operation expressed in IDL (as you'd expect).
> Especially true for multiplies, but everything sees a healthy speed-up.
> It even takes medians faster than median().
>
> The thing I don't like about REDUCE:
>
> It is horribly ugly.
>
> The reason it is horribly ugly is all those damn types. If you followed
> Ronn Kling's book, you'd know he recommends handling multiple types
> like:
>
> switch(type){
> case IDL_TYP_INT: myvar=(short *) foo; stuff1; stuff2; stuff3; break;
> case IDL_TYP_LONG: myvar=(int *) foo; stuff1; stuff2; stuff3; break;
> case IDL_TYP_FLOAT: myvar=(float *) foo; stuff1; stuff2; stuff3;
> break;
> ...

> ...
> }
>
> That is, just replicate things over and over again for the various
> types. REDUCE works natively in 9 types. Luckily, I didn't have to
> copy everything over nine times as above, but in essence that's what I
> did. I just used a host of clever C pre-processor directives to
> indirect the type replication.
>
> OK, no problem. But what happens is "stuff" is large. For example,
> finding a median takes about 75 lines of code with all the
> initialization etc. What's more you need a separate copy of the same
> code not just for the 9 types, but also for the cases in which you're
> possibly promoting to double, or float. A given piece of code can end
> up being replicated 18 times, with slight differences like:
>
> float *p=IDL_MakeTempArray(IDL_TYP_FLOAT,...);
>
> vs.
>
> short *p=IDL_MakeTempArray(IDL_TYP_INT,...);
>
> vs.
>
> int *p=IDL_MakeTempArray(IDL_TYP_LONG,...);
>
> and so on, ad infinitum.
>
> For a couple lines of code, this isn't too bad, but when you're forced
> to shoehorn a 70 line function into a macro just to replicate it 9 or 18
> times with some very subtle change, it gets ugly, and bloated. My
> nested loops which do the magic of threading the calculations occur 81
> times in the code, after pre-processing! Yuck.
>
> If this is how IDL handles dealing with multiple types internally, well
> that makes me very sad (and allows me to understand why their median
> only deals with two types).
>
> The question, to all you C-programmers: is there a better way?
>
> In order to phrase the challenge more sensibly, consider a function that
> will take the maximum of an array of data:
>
> IDL_LONG maximum(data)
>
> The catch is data will be of whatever numeric type the user likes (see
> the list in external/export.h under the IDL directory for a list of
> them).

>
> First recognize that the code logic to compute the maximum will be the
> same, both symbolically for all types (e.g. "if data[i]>max then
> max=data[i]"), and for many types, in the compiled code itself. Can you
> come up with a portable way to write and call maximum() which avoids any
> of the repitition intrinsic in the straightword approach, that is, to
> avoid compiling in the code like
>
> "if data[i]>max..."
>
> once for each type?
>
> Thought I'd give it a shot. I'll release REDUCE to the masses once I
> sort these issues out.
>
> JD
>
>

Subject: Re: REDUCE
Posted by John-David T. Smith on Fri, 30 Mar 2001 05:32:17 GMT
View Forum Message <> Reply to Message

Kenneth Mankoff wrote:
>
>> The question, to all you C-programmers: is there a better way?
> [snip]
>> ...the code logic to compute the maximum will be the same, both
>> symbolically for all types for many types, in the compiled code itself.
>
> Hi JD,
>
> hmmm... not 100% sure, but wouldn't c++ templates solve this problem?
>
> And for the cases where it is "symbolically" the same but not "compiled
> the same", I'm not sure what this means, but I'm guessing you would handle
> these cases with overloading your operators.
>
> Of course, C isn't C++, so this might not help.
>
> I can provide code examples and more info if you wish.

Thanks for the suggestion. I had thought of that option, but I don't know much about templates, nor about linking C++ to IDL. I wonder whether the templates are just similar to my super macro for creating a different version for each type. Can you frame the maximum function I suggested in terms of a skeleton template which would operate on all the data types?

My comment with respect to compiled and symbolic maybe wasn't clear. I really just meant that you have this same code replicated over and over, with minor changes in the types of the variables used, but otherwise logically and symbolically intact. I can imagine the compiler emitting different code for, e.g., multiplying two integers, vs. two floats, but I can also imagine other types where the codes emitted are exactly the same. Obviously, you can't get something for nothing, but if real repition exists within the compiled code, you should be able to eliminate it somehow.

Thanks again,

JD

Subject: Re: REDUCE
Posted by Kenneth Mankoff on Fri, 30 Mar 2001 07:30:45 GMT
View Forum Message <> Reply to Message

On Fri, 30 Mar 2001, John-David Smith wrote:

> Kenneth Mankoff wrote:
>>
>>> The question, to all you C-programmers: is there a better way?
>> [snip]
>>> ...the code logic to compute the maximum will be the same, both
>>> symbolically for all types for many types, in the compiled code itself.
>>
>> Hi JD,
>>
>> hmmm... not 100% sure, but wouldn't c++ templates solve this problem?
>>
>> And for the cases where it is "symbolically" the same but not "compiled
>> the same", I'm not sure what this means, but I'm guessing you would handle
>> these cases with overloading your operators.
>>
>> Of course, C isn't C++, so this might not help.
>>
>> I can provide code examples and more info if you wish.
>

> Thanks for the suggestion. I had thought of that option, but I don't
> know much about templates, nor about linking C++ to IDL.

I think you can just put the regular C code in there to connect to the client. Sorta like writing C+.

> I wonder whether the templates are just similar to my super macro for
> creating a different version for each type. Can you frame the maximum
> function I suggested in terms of a skeleton template which would operate
> on all the data types?

```
// function definition
template<class idlType> idlType maximum( idlType v0, idlType v1 )
{
   if ( v0 > v1 )
      return v0;
   return v1;
}

// function instantiation
int main(void)
{
   float f = maximum( 4.2, 4.3 );
   int   i = maximum( 43, 42 );
}
```

I thought for "symbolically" and "logically" the same, you were referring
to comparison of strings or objects or structs or other types, where the
standard ">" won't work, and you need "strcmp()" or something else.

-k.

 -- ---------
   Kenneth Mankoff   LASP
mankoff@lasp.colorado.edu   1234 Innovation Drive
     303.492.3264   Boulder, CO 80303
 -- ---------

Subject: Re: REDUCE
Posted by Richard Younger on Sun, 01 Apr 2001 18:55:31 GMT
View Forum Message <> Reply to Message

John-David Smith wrote:
>
> Kenneth Mankoff wrote:
>>
>>> The question, to all you C-programmers: is there a better way?
>> [snip]
>>> ...the code logic to compute the maximum will be the same, both
>>> symbolically for all types for many types, in the compiled code itself.
>>
>> Hi JD,

>>
>> hmmm... not 100% sure, but wouldn't c++ templates solve this problem?
>>
>> And for the cases where it is "symbolically" the same but not "compiled
>> the same", I'm not sure what this means, but I'm guessing you would handle
>> these cases with overloading your operators.
>>
>> Of course, C isn't C++, so this might not help.
>>
>> I can provide code examples and more info if you wish.
>
> Thanks for the suggestion. I had thought of that option, but I don't know much
> about templates, nor about linking C++ to IDL. I wonder whether the templates
> are just similar to my super macro for creating a different version for each
> type. Can you frame the maximum function I suggested in terms of a skeleton
> template which would operate on all the data types?
>
> My comment with respect to compiled and symbolic maybe wasn't clear. I really
> just meant that you have this same code replicated over and over, with minor
> changes in the types of the variables used, but otherwise logically and
> symbolically intact. I can imagine the compiler emitting different code for,
> e.g., multiplying two integers, vs. two floats, but I can also imagine other
> types where the codes emitted are exactly the same. Obviously, you can't get
> something for nothing, but if real repition exists within the compiled code, you
> should be able to eliminate it somehow.
>
> Thanks again,
>
> JD

Hi JD and Ken,

I agree with Ken that the most obvious and easiest C++ solution is templates and operator overloading. I have a little experience DLMing with C++, and it works just fine. The calling conventions of C and C++ can be set exactly the same, so the limitations are exactly the same.

What is the distinction between the different cases? Are you primarily worried with arithmetic, indirection, or member changes. e.g:

```
(float a * 2)  vs  (int a * 2) or
(float*)a  vs  (int*)a  or
2*value.f  vs  2*value.i
```

The overloading and template solutions work well on the first two problems, and not well at all on the last category, because AFAIK there's no good, compact way to make run-time distinctions with members. It's because different explicit symbols are used, as opposed

to different implicit types. You end up using lots of switch - case statements. I suppose you could put the switch into an operator to extract the value of a data element, but then you end up switching every time you access an array element, instead of once at the beginning. I'd think it would be slower than your super-macro. Maybe someone else knows a better solution.

And I know that IDL uses unions in its variable definitions, so I suspect that you are running into the third problem.

As for using a nifty silver bullet C++ feature, the use of unions is generally discouraged in C++ for the above reasons. I suppose you could go heavy-duty C++ and start throwing around wrapper classes and Standard Template Library iterators, but I'm just barely familiar with that stuff (iterators anyway), and don't know enough to really know if that solution is simple or fast or not. Besides, I don't think JD is that interested in learning lots of C++ language semantics for limited return.

On a side note, Your REDUCE package seems to be very similar to a feature that I really would like RSI to implement; namely, Einstein-summation or dummy-index notation. Something that would result in operations like

```
epsilon = fltarr(3, 6, 9)
E_one = fltarr(9)
E_two = fltarr(6)

epsilon[%1, %2, %3]*E_one[%3]*E_two[%2]
```

would multiply the elements of epsilon by E_one on the corresponding (3rd) index and sum over that index. Especially for those of us working with lots of fields in tensor notation, it would save lots of for loops, and I'm sure that a built-in facility would save time over the for loop.

Has anyone else found themselves yearning for such a feature?

Rich

--
Richard Younger
Assistant Technical Staff MIT Lincoln Laboratory
Electro-Optical Materials and Devices Mail Stop C-130
Email: younger@ll.mit.spmdecoy.edu 244 Wood St.
Phone: (781)981-4464 Lexington, MA 02144-9108

(My opinions on this forum are not endorsed, warrantied, etc. by my

employer.)

Subject: Re: REDUCE
Posted by John-David T. Smith on Sun, 01 Apr 2001 22:01:19 GMT
View Forum Message <> Reply to Message

Richard Younger wrote:
>
> John-David Smith wrote:
>>
>> Kenneth Mankoff wrote:
>>>
>>>> The question, to all you C-programmers: is there a better way?
>>> [snip]
>>>> ...the code logic to compute the maximum will be the same, both
>>>> symbolically for all types for many types, in the compiled code itself.
>>>
>>> Hi JD,
>>>
>>> hmmm... not 100% sure, but wouldn't c++ templates solve this problem?
>>>
>>> And for the cases where it is "symbolically" the same but not "compiled
>>> the same", I'm not sure what this means, but I'm guessing you would handle
>>> these cases with overloading your operators.
>>>
>>> Of course, C isn't C++, so this might not help.
>>>
>>> I can provide code examples and more info if you wish.
>>
>> Thanks for the suggestion. I had thought of that option, but I don't know much
>> about templates, nor about linking C++ to IDL. I wonder whether the templates
>> are just similar to my super macro for creating a different version for each
>> type. Can you frame the maximum function I suggested in terms of a skeleton
>> template which would operate on all the data types?
>>
>> My comment with respect to compiled and symbolic maybe wasn't clear. I really
>> just meant that you have this same code replicated over and over, with minor
>> changes in the types of the variables used, but otherwise logically and
>> symbolically intact. I can imagine the compiler emitting different code for,
>> e.g., multiplying two integers, vs. two floats, but I can also imagine other
>> types where the codes emitted are exactly the same. Obviously, you can't get
>> something for nothing, but if real repition exists within the compiled code, you
>> should be able to eliminate it somehow.
>>
>> Thanks again,
>>
>> JD

>
> Hi JD and Ken,
>
> I agree with Ken that the most obvious and easiest C++ solution is
> templates and operator overloading. I have a little experience DLMing
> with C++, and it works just fine. The calling conventions of C and C++
> can be set exactly the same, so the limitations are exactly the same.
>
> What is the distinction between the different cases? Are you primarily
> worried with arithmetic, indirection, or member changes. e.g:
>
> (float a * 2) vs (int a * 2) or
> (float*)a vs (int*)a or
> 2*value.f vs 2*value.i
>
> The overloading and template solutions work well on the first two
> problems, and not well at all on the last category, because AFAIK
> there's no good, compact way to make run-time distinctions with
> members. It's because different explicit symbols are used, as opposed
> to different implicit types. You end up using lots of switch - case
> statements. I suppose you could put the switch into an operator to
> extract the value of a data element, but then you end up switching every
> time you access an array element, instead of once at the beginning. I'd
> think it would be slower than your super-macro. Maybe someone else
> knows a better solution.

The first two things are what I'm concerned about, with indirection
modified to include declaration. Here's an example of post-preprocessor
code for threading the "max" operation (cleaned up a fair bit):

```
if(maxQ) {switch( type ) {
 case IDL_TYP_BYTE:
  {
    UCHAR *tin,*tout,tmp;
    tout=( UCHAR *)out;
    tin=( UCHAR *)arg[0]->value.arr->data;
    for(i=0,base=0;i<new_nel;base+=skip) {
      for(j=0;j<atom;j++) {
  tmp=tin[j+base];
  for(ind=j+base;ind<j+base+atom*n_cdim;ind+=atom) {
   if(tin[ind]>tmp)tmp=tin[ind];
  }
  tout[i++]=tmp;
      }
    }
  }
  break;
 case IDL_TYP_INT:
```

```
  {
    short *tin,*tout,tmp;
    tout=( short *)out;
    tin =( short *)arg[0]->value.arr->data ;
    for(i=0,base=0;i<new_nel;base+=skip) {
      for(j=0;j<atom;j++) {
 tmp=tin[j+base];
 for(ind=j+base;ind<j+base+atom*n_cdim;ind+=atom) {
  if(tin[ind]>tmp)tmp=tin[ind];
 }
 tout[i++]=tmp;
      }
    }
  }
  break;
case IDL_TYP_LONG: {
 ....
```

And it goes on and on for all 9 types. "out" is the result of an IDL_MakeTempArray() call, and needs to be cast correctly, as does the input array data. A tmp variable of the correct size is also initialized. This is the only difference in all 9 version of the generated code. Granted, this is a very simple example, but what I am looking for is a solution which makes use of the redundancy in this code to avoid generating most of it. I may be asking more out of compilers than they can offer.

I think what C++ templates would do is basically the same thing I'm doing, but in a much cleaner way (i.e. not using ugly nested macros). That is, it would "instantiate" a different version of my looping max-finding function 9 times, and the code would bloat just as much. This isn't a big deal for this little function, but imagine a very large template function being duplicated 9 (or 18) times. I'm beginning to suspect there's no real way around this.

> On a side note, Your REDUCE package seems to be very similar to a
> feature that I really would like RSI to implement; namely,
> Einstein-summation or dummy-index notation. Something that would result
> in operations like
>
> epsilon = fltarr(3, 6, 9)
> E_one = fltarr(9)
> E_two = fltarr(6)
>
> epsilon[%1, %2, %3]*E_one[%3]*E_two[%2]
>
> would multiply the elements of epsilon by E_one on the corresponding

> (3rd) index and sum over that index. Especially for those of us working
> with lots of fields in tensor notation, it would save lots of for loops,
> and I'm sure that a built-in facility would save time over the for loop.
>

I.e. an implicit double sum over the second and third indices? I
wouldn't do this with for loops. I would use rebin, and total, like:

```
res=total(total(epsilon*rebin(reform(e_one,1,1,s[2]),s)*  $
        rebin(1#e_two,s),1),2)
```

Admittedly, your notation is somewhat cleaner. If these are confusing,
see my tutorial (to be posted) concerning rebin+reform in action.

JD

Subject: Re: REDUCE
Posted by Richard Younger on Mon, 02 Apr 2001 03:31:47 GMT
View Forum Message <> Reply to Message

JD Smith wrote:
>
> Richard Younger wrote:
>>
>> John-David Smith wrote:
>>>
>>> Kenneth Mankoff wrote:
>>>>

[SNIP]

> generated code. Granted, this is a very simple example, but what I am
> looking for is a solution which makes use of the redundancy in this code
> to avoid generating most of it. I may be asking more out of compilers
> than they can offer.

I was thinking about this a little more, and then it hit me: You
probably don't want templates.

Templates are a means for providing ~compile time~ polymorphism. With
compile time polymorphism, you'll always need some sort of switch()
statement or a decision table to decide which compiled path to take at
runtime. What you're really looking for is ~run time~ polymorphism.

Run time polymorphism is provided for with function pointers in C and virtual functions (and parent class / derived class hierarchies) in C++. Now I suppose I could try to cook up a C++ example using virtuality, but outside of a brief exercise or two, I've never actually used it. So here's an example of a function pointer table. Tell me if you like it. No C++ required.

WARNING -- large code block follows -- see bottom for more actual text.

```
/* myheaderfile.h ---------------------------------------*/

#include <stdio.h>
#include "export.h"

void NULL_process (char* out, IDL_VARIABLE* arg, int new_nel, int skip,
int atom, int n_cdim )
{
 /*IDL_MESSAGE(... "She canna handle this type, cap'n!" ... )*/
 return;
}

/* or use a macro to generate these in the header file */
void BYTE_process (char* out, IDL_VARIABLE* arg, int new_nel, int skip,
int atom, int n_cdim )
{
 int i, j, ind, base;

     UCHAR *tin,*tout,tmp;
     tout=( UCHAR *)out;
     tin=( UCHAR *)arg->value.arr->data;
     for(i=0,base=0; i<new_nel; base+=skip) {
       for(j=0;j<atom;j++) {
         tmp=tin[j+base];
         for(ind=j+base; ind<j+base+atom*n_cdim; ind+=atom) {
           if(tin[ind]>tmp)
      tmp=tin[ind];
         }
         tout[i++]=tmp;
       }
     }
 return;
}

void INT_process (char* out, IDL_VARIABLE* arg, int new_nel, int skip,
int atom, int n_cdim )
{
 int i, j, ind, base;
     short *tin,*tout,tmp;
```

```
    tout=( short *)out;
    tin =( short *)arg->value.arr->data ;
    for(i=0,base=0;i<new_nel;base+=skip) {
      for(j=0;j<atom;j++) {
        tmp=tin[j+base];
        for(ind=j+base;ind<j+base+atom*n_cdim;ind+=atom) {
          if(tin[ind]>tmp)tmp=tin[ind];
        }
        tout[i++]=tmp;
      }
    }
}

/* etc ... */
/*----end of header-------------------------------------------------*/

/*---start of main---------------*/
#include "myheaderfile.h"

/* typedef pointer to processing function */
typedef void (*PROC_FUNC) (char* , IDL_VARIABLE*, int, int, int, int);

void main (void)
{
 PROC_FUNC process_table[] = {
  NULL_process,
  BYTE_process,
  INT_process
  /* etc ...
   to fill out this table, you'll need to look at
   the macro IDL_TYP_ values in export.h and put in
   placeholders for those types that you can't handle.*/
 };

 /*RSI will scowl at this method, because the table relies on
   the values of the macro-defined type indices (e.g. #define
IDL_TYP_INT 2)
   staying the same, which they do not guarantee.
 */

 /*dummy arguments*/
 char* out = NULL;
 int new_nel=0, skip=0, atom=0, n_cdim=0;
 IDL_VARIABLE* arg[5];
 int type = 1;

 process_table[type](out, arg[0], new_nel, skip, atom, n_cdim );
```

```
 return;
}
/*----------------------*/
```
... now I ~know~ this compiles, and will run the relevant functions (I tested with print statements), but I may have mangled your math. Double check before cut and pasting.

You could also stick the typedef and the table in the header, making it global, if that suited your sense of style better.

for completeness, here's a template function if you want to take a look at it. Notice it still requires the switch, because there must be an argument of the templated type.

```
#include <stdio.h>
#include "export.h"


template <class Item>
void process (Item tmp, IDL_VARIABLE* arg[], void* out, int new_nel, int
skip, int atom, int n_cdim )
{
 int i, j, ind, base;

 Item *tin, *tout;
    tout=( Item *)out;
     tin =( Item *)arg[0]->value.arr->data ;
     for(i=0,base=0; i<new_nel; base+=skip) {
       for(j=0; j<atom; j++) {
         tmp=tin[j+base];
         for(ind=j+base; ind<j+base+atom*n_cdim; ind+=atom) {
           if(tin[ind]>tmp)tmp=tin[ind];
         }
         tout[i++]=tmp;
       }
     }
  return;
}

void main(void)
{
 int type = 1;

 void* out = NULL;
 int new_nel=0, skip=0, atom=0, n_cdim=0;
```

```
 IDL_VARIABLE* arg[5];

 switch( type ) {
 case IDL_TYP_BYTE: {
  UCHAR tmp = 0;
  process (tmp, arg, out, new_nel, skip, atom, n_cdim );
  }
  break;
 case IDL_TYP_INT: {
  short tmp = 0;
  process (tmp, arg, out, new_nel, skip, atom, n_cdim );
  }
  break;
 //etc...
 }

 return;
}
```

> I.e. an implicit double sum over the second and third indices? I
> wouldn't do this with for loops. I would use rebin, and total, like:
>
> res=total(total(epsilon*rebin(reform(e_one,1,1,s[2]),s)* $
> rebin(1#e_two,s),1),2)
>
> Admittedly, your notation is somewhat cleaner. If these are confusing,
> see my tutorial (to be posted) concerning rebin+reform in action.
>
> JD

I use reform commonly, but I'll have to side with David and say that the use of REBIN and HISTOGRAM for anything but their most obvious uses is a black art that intimidates me. *shudder* :-) Hey, wait a second! You just want to put together this REDUCE package so you can put REBIN and HISTOGRAM to more arcane uses than before, don't you?

My notation also allows almost direct copying of formulas from a large number of textbooks, which is one major reason I'm partial to it. Unfortunately, it's wishful thinking, and REBIN is not. :-)

Rich

--
Richard Younger
Assistant Technical Staff MIT Lincoln Laboratory
Electro-Optical Materials and Devices Mail Stop C-130

Email: younger@ll.mit.spmdecoy.edu 244 Wood St.
Phone: (781)981-4464 Lexington, MA 02144-9108

Subject: Re: REDUCE
Posted by John-David T. Smith on Mon, 02 Apr 2001 18:17:40 GMT
View Forum Message <> Reply to Message

Richard Younger wrote:
>
>
> [SNIP]
>
>> generated code. Granted, this is a very simple example, but what I am
>> looking for is a solution which makes use of the redundancy in this code
>> to avoid generating most of it. I may be asking more out of compilers
>> than they can offer.
>
>
>
> I was thinking about this a little more, and then it hit me: You
> probably don't want templates.
>
> Templates are a means for providing ~compile time~ polymorphism. With
> compile time polymorphism, you'll always need some sort of switch()
> statement or a decision table to decide which compiled path to take at
> runtime. What you're really looking for is ~run time~ polymorphism.
>
> Run time polymorphism is provided for with function pointers in C and
> virtual functions (and parent class / derived class hierarchies) in
> C++. Now I suppose I could try to cook up a C++ example using
> virtuality, but outside of a brief exercise or two, I've never actually
> used it. So here's an example of a function pointer table. Tell me if
> you like it. No C++ required.
>
> WARNING -- large code block follows -- see bottom for more actual text.

Hi Rich:

Thanks again for the ideas. Unfortunately, this implements, albeit in a more manageable and less ugly way, pretty much what I'd already accomplished with macros. That is, each and every of the functions BYTE_foo, INT_foo, ULONG_foo, etc., get compiled and included in the executeable separately, and you essentially choose among them with the run-time type information. This works, but leads to extreme code bloat

if you're replicating a large function 10's of times.

So far, all these solutions have succeeded in allowing the program to branch to some other portion of its code, depending on the type. I was looking for a solution in which the executeable could use the exact same piece of compiled code to accomplish the calculation with a variety of types. That is, run-time, *in-place* type polymorphism. One sneaky way of reducing this duplication is to perform in place casting only where you need it. Here's an example which you can try, in which the variable "tmp" is used in many ways:

```
#include <stdlib.h>
#include <stdio.h>
#include "/usr/local/rsi/idl/external/export.h"

main() {
  int a=5;
  float b=6.;
  IDL_ULONG64 c=1000000;
  void *tmp=malloc(sizeof(IDL_ULONG64));

  *(int *)tmp=a;

  printf("GOT %d\n",*(int *)tmp);

  *(float *)tmp=b;
  printf("GOT %f\n",*(float *)tmp);

  *(IDL_ULONG64 *)tmp=c;
  printf("GOT %Lu\n",*(IDL_ULONG64 *)tmp);
}
```

Now, all the code emitted by the compiler which, for example, performs the loops, can be the same. All that's required is branched code for casting your input and output arrays, and any statement which uses that cast data. I.e, you could have something like:

```
void ALL_process (void *in, void *out, int skip, int atom, int n_cdim,
    int new_nel )
{
  IDL_LONG i, j, ind, base;
  void *tmp=(void *)IDL_GetScratch(0,1,sizeof(IDL_ULONG64));
  for(i=0,base=0;i<new_nel;base+=skip) {
    for(j=0;j<atom;j++) {
      switch(type) {
      case IDL_TYP_BYTE:
 *(UCHAR *)tmp=((UCHAR *)in)[j+base];
 break;
```

```
      case IDL_TYPE_INT:
 *(short *)tmp=((short *)in)[j+base];
 break;
       .... (etc.)
      }
      for(ind=j+base;ind<j+base+atom*n_cdim;ind+=atom) {
 switch(type) {
 case IDL_TYP_BYTE:
  if( ((UCHAR *)in)[ind]>*(UCHAR *)tmp )
    *(UCHAR *)tmp=((UCHAR *)in)[ind];
  break;
 case IDL_TYPE_INT:
  if( ((short *)in)[ind]>*(short *)tmp )
    *(short *)tmp=((short *)in)[ind];
  break;
       .... (etc.)
 }
      }
      switch(type) {
      case IDL_TYP_BYTE:
 ((UCHAR *)out)[i++]=*(UCHAR *)tmp;
 break;
      case IDL_TYPE_INT:
 ((short *)out)[i++]=*(short *)tmp;
 break;
      .... (etc.)
      }
   }
  }
}
```

I guess it's hard to say how much of a bloat savings this is, and whether it impacts performance. Obviously, you'd need a clever way to do all those run-time branched type castings, rather than by hand. Maybe a perl script would be easier than trying to futz with the pre-processor (but less portable).

Anyway, let me know if you think of anything along these lines.

Thanks again,

JD

Subject: Re: REDUCE
Posted by Martin Schultz on Tue, 03 Apr 2001 08:08:51 GMT
View Forum Message <> Reply to Message

JD Smith wrote:
>
> ... Unfortunately, this implements, albeit in a
> more manageable and less ugly way, pretty much what I'd already
> accomplished with macros. That is, each and every of the functions
> BYTE_foo, INT_foo, ULONG_foo, etc., get compiled and included in the
> executeable separately, and you essentially choose among them with the
> run-time type information. This works, but leads to extreme code bloat
> if you're replicating a large function 10's of times.

Unless RSI has a Wonderland solution, they probably do the same.
Just a "reverse" idea: Looking at one of your recent examples, it
seemed as if all the code but the definition of the out array was type
independent. So, why not put that code in a macro? Something like

```
if(maxQ) {switch( type ) {
 case IDL_TYP_BYTE:
   {
     UCHAR *tin,*tout,tmp;
     tout=( UCHAR *)out;
     tin=( UCHAR *)arg[0]->value.arr->data;
#include max_fun.h
     }
   }
   break;
 case IDL_TYP_INT:
   {
     short *tin,*tout,tmp;
     tout=( short *)out;
     tin =( short *)arg[0]->value.arr->data ;
#include max_fun.h
     }
   }
   break;
...
```

Doesn't solve the problem but may make the code easier to read and
maintain (just imagine you want to fix a bug in 10 copies of almost
the same code)

Cheers,

Martin

--
 [[[[[[[[[[
[[Dr. Martin Schultz   Max-Planck-Institut fuer Meteorologie   [[

[[Bundesstr. 55, 20146 Hamburg [[
[[phone: +49 40 41173-308 [[
[[fax: +49 40 41173-298 [[
[[martin.schultz@dkrz.de [[
[[[[[[[[[[