Subject: Re: IDL v5.1 impressions
Posted by J.D. Smith on Thu, 14 May 1998 07:00:00 GMT
View Forum Message <> Reply to Message

Mr. Bahrami,

I'm taking the liberty of cross-posting to the idl-pvwave newsgroup my reply to your prompt and informative response to my previous posting. Your explanation goes a long way toward appeasing my discontent with the by-reference inherited-keyword mechanism just introduced, and I think it explains the mechanism much more clearly than the IDL reference material, and would therefore benefit the general IDL community. There are still a few issues which are unclear to me, however.

> Mr Smith,
>
> (Forgive me if it's Dr. --- your signature doesn't make it clear)

Just 1 more year! (so I've said for the previous few...)

>
> Thanks for the nice words about IDL 5.1.
>
> You raised some interesting points about the new keyword inheritance
> mechanism. I'll try to answer them...
>
>
>> But having said that I have to bring down the joy level and remark that,
>> although I applaud RSI for implementing pass by reference inherited
>> keywords, I don't understand why they did it in such a complicated and
>> non-intuitive way! Having two keyword inheritence mechanisms seems
>> unnecessary. The manual quotes:
>
> It would be unnecessary if we were free of the need for backwards
> compatability. However, I know of many applications that build their
> own _EXTRA structures and/or modify the one they're passed before they pass
> it on to the inheriting routine. Hence, the old behavior must continue
> to be supported. I was unable to see how a simple reliable by-reference
> mechanism could be grafted onto the struct based _EXTRA mechanism. Given
> the need to preserve the old behavior, my modified goals became:
>
> - Make the new mechanism similar enough to the old one that
> people used to the old one can rapidly understand it.
>
> - Make it easy to switch from the old one to the new one.
>
> I'm especially pleased with the second point --- you can change the
> method used by modifiying only the signature of the routine itself. There

> is no need to hunt down and replace all the calls to the routine with
> a different syntax.

I agree this second point is critical, and of course I understand the pressing need for backwards compatibility. I am obviously not privy to the internals of IDL, but it had seemed obvious to me that both conditions could be satisfied with only 1 mechanism (the same _EXTRA that we're all used to)... more on this later.

>
>> **********
>> By contrast, the "pass by value" (_EXTRA) keyword inheritance mechanism
>> is useful in the following situations:
>>
>> 1. Your routine needs access to the values of the extra keywords for
>> some reason.
>>
>>
>> 2. You want to ensure that variables specified as keyword parameters are
>> not changed by a called routine.
>> **********
>>
>> Number 1 is silly. Of course I might need the values along the way!
>
> Just to be clear, "your routine" is refering to the intermediate routine.
> There is a caller, the intermediate inbetween routine, and the ultimate
> called routine that gets the inherited keywords. (I think you understood
> this, I mention it just in case...)
>
> If you need the values along the way, then you probably know what they
> are. In this case, use real keywords instead of inheritance and you
> have full access.
>
>> When I give a positional argument to a procedure, like:
>>
>> myfunc, array
>>
>> I both expect to have access to the value of array, and to have any
>> changes I make to array be available to the calling environment. That
>> is the whole beauty of pass by reference. Passing by reference
>> *doesn't* mean passing without value! An example of when you might need
>> the values in the set of extra parameters if for general pupose error
>> checking; suppose your procedure would only allow keywords with floats,
>> no matter what keyword it is, for passing on up to some other
>> procedure. We don't want to define all possible keywords, we just want
>> to ensure that for any passed containing a value or a variable defined
>> with a value, that value is a float.
>

>
> Well, OK. But if they were inherited *and* visible in the intermediate routine,
> then they will have to have local variable names inside the intermediate
> routine. What variable name should be used? This is the tricky point, because
> the signature of the routine doesn't say anything about it. If you give it
> the name it had in the calling routine or the name of the keyword, then you
> are at risk of a name collision with a previously existing local variable that
> is not intended to be visible outside the routine, and which is completely
> unrelated to the keyword being inherited. Obviously, it is a bad
> idea to allow the caller to determine what variables are local and which are
> not! Another alternative is to give them some randomly generated name.
> This didn't seem like an especially usable answer. Pass them in a heap
> variable? That seems expensive and difficult to manage.

The solution which has worked for the normal _EXTRA passing mechanism
would seem appropriate... simply make it a structure of the name
specified in the procedure definition. This obviously implies making an
extra copy of inherited keywords and their values inside an intermediate
routine which may not, in the end, even use these values, and therefore
suffers the same inefficiency as the standard _EXTRA mechanism.

>
> I finally came to the conclusion that if a routine cares about a given
> keyword, the programmer knows enough about it to just declare the
> keyword and not use inheritance. Use of inheritance means that the
> intermediate routine is not involved, hence should not expect to
> access it. Hence, the simplest thing to do is not
> allow the variable to be visible in the intermediate level. This solves
> the name-clash problem.
>
> Your point about type checking is true, but should be done by the actual
> inheriting routine, where the program knows everything it needs. After
> all, the whole point of keyword inheritance is to allow the caller to
> pass things "through" an intermediary without the intermediary being
> involved. For example, a wrapper to PLOT, where the caller can pass
> PLOT keywords without the wrapper having to be aware of them.
>
>
>> And as for Number 2... why don't we have different mechanisms for
>> positional argument passing, then? If I pass an array to myfunc, I have
>> no way of ensuring that array isn't modified, but I don't care, since I
>> know what myfunc does and act accordingly.
>>
>> I have been happy with the choice of reference vs. value passing for the
>> positional arguments. We are all used to the rules. Why didn't RSI
>> simply extend these same rules to include keywords? I think the answer
>> might be that this was a much harder problem to code...
>

> No, not harder to code --- the backwards compatability is the issue.
> Regular keywords *do* use the standard rules, and the _REF_EXTRA code
> which also implement the standard rules) is simpler than the _EXTRA code
> (no structs to build and deconstruct). _REF_EXTRA is also more efficent
> at runtime.
>

Of course what I was implying is the extension of the "regular rules" to
inherited keywords. It is evident to me now that it would have been
possible to implement a single, full by-reference model in both the
intermediate and final called routine by simply extending the _EXTRA
structure definition somewhat. If a passed inherited keyword value is
by-reference (according to the "regular" rules), then it would be tied
to a shared location, and mapped into the finally-called routine's data
space (just like _REF_EXTRA does). The "value" would still be available
via the regular KEYWORD/VALUE pair structure in the intermediate
routine, but would be a different entity for non-reference vs. reference
parameters (the latter would actually just access the value in the
shared location, allowing by-reference in the intermediate routine,
too!). That would make things consistent with the standard rules,
preserve backwards compatibility (the _EXTRA struct would still have
keyword/value pairs as before), and would be invisible. The only
possible complication to backwards compatibility is if people were
counting on the fact that inherited keywords are passed by value
only.... It would also require special-case programming for _EXTRA
structures vs. regular structures.

But, having said all this, I must point out that this possible solution
would probably not be as efficient as the current _REF_EXTRA mechanism
(though it does more at once). I am reasonably satisfied with the new
solution, but only because I've used inherited keywords almost
exclusively as you've envisioned their ideal use: as intermediate
entitities whose values we shouldn't care about (except in the
ultimately called routine). If this were their only use, you could have
abandoned the old _EXTRA altogether. But, as we both know, there is
lots of code out there which uses keyword inheritence in an altogether
different way, for convenience mainly, examining the values along the
way. So it boils down to a matter of programming hygiene, really:
either you stop relying on having those values, or you don't get
by-reference passing. I think I can live with that, and I hope everyone
else can too.

> The original _EXTRA mechanism was designed to solve a simpler problem.
> At the time, it's shortcomings were not apparent. In hindsight, it would
> have been better to do it the _REF_EXTRA way and not have 2 mechanisms,
> but design error are usually easier to see in hindsight.
>
> The manual is simply trying to show that there are some attributes of

> _EXTRA (given that it does exist) that might be useful, not to prove
> that 2 inheritance mechanisms are necessary.
>
>
>> And one more problem. The ever useful function arg_present() chokes
>> when confronted with _ref_extra keywords.... let's make an example,
>> using the by reference mechanism that RSI has provided us:
>
> ... <code omitted>
>
>> IDL> testrefext, var=10
>> RE STRING = Array[1]
>> V INT = 10
>> Arg present: 1
>>
>> 10 is a valid variable for passing back? I don't think so.
>
> Neither do I. You've discovered a bug. Interestingly, this managed to get
> through 3 beta test cycles without being discovered. I believe this is because
> ARG_PRESENT always says TRUE in this case. This means that sometimes a
> routine will do more work than it might have otherwise.
>
> If it always said FALSE, it would have broken code (instead of simply allowing
> them to do more work than necessary). This would have been caught quickly.
>
> I've entered it into our bug tracking system to be sure it gets fixed
> for the next release. Thanks!
>
> - Ali Bahrami, RSI

I'm glad to have rooted out another bug. Seems that with all the bugs/design issues I've found in shipping versions of IDL (object procedure argument checking, obj_isa() on object arrays, by-reference keyword inheritence requirement for full OOP functionality, arg_present() bugs and the need for it in the first place, ...), I might deserve an IDL coffee mug :). Thanks again for giving all these matters such prompt attention. It's good to know you at RSI regard your users so highly.

JD

--
J.D. Smith |*| WORK: (607) 255-5842
Cornell University Dept. of Astronomy |*| (607) 255-4083
206 Space Sciences Bldg. |*| FAX: (607) 255-5875
Ithaca, NY 14853 |*|