NAME

Devel::CallChecker – custom op checking attached to subroutines

SYNOPSIS

```
# to generate header prior to XS compilation

perl -MDevel::CallChecker=callchecker0_h \
    -e 'print callchecker0_h' > callchecker0.h

# in Perl part of module

use Devel::CallChecker;

/* in XS */

#include "callchecker0.h"

cv_get_call_checker(cv, &ckfun, &ckobj);
static OP *my_ckfun(pTHX_ OP *o, GV *namegv, SV *ckobj);
cv_set_call_checker(cv, my_ckfun, ckobj);
```

DESCRIPTION

This module makes some new features of the Perl 5.14.0 C API available to XS modules running on older versions of Perl. The features are centred around the function `cv_set_call_checker`, which allows XS code to attach a magical annotation to a Perl subroutine, resulting in resolvable calls to that subroutine being mutated at compile time by arbitrary C code. This module makes `cv_set_call_checker` and several supporting functions available. (It is possible to achieve the effect of `cv_set_call_checker` from XS code on much earlier Perl versions, but it is painful to achieve without the centralised facility.)

This module provides the implementation of the functions at runtime (on Perls where they are not provided by the core). It also, at compile time, supplies the C header file and link library which provide access to the functions. In normal use, ‘‘callchecker0_h’’ and ‘‘callchecker_linkable’’ should be called at build time (not authoring time) for the module that wishes to use the C functions.

CONSTANTS

callchecker0_h

Content of a C header file, intended to be named "`callchecker0.h`". It is to be included in XS code, and `perl.h` must be included first. When the XS module is loaded at runtime, the `Devel::CallChecker` module must be loaded first. This will result in the Perl API functions `rv2cv_op_cv`, `ck_entersub_args_list`, `ck_entersub_args_proto`, `ck_entersub_args_proto_or_list`, `cv_get_call_checker`, and `cv_set_call_checker`, as defined below and in the Perl 5.14.0 API, being available to the XS code.

callchecker_linkable

List of names of files that must be used as additional objects when linking an XS module that uses the C functions supplied by this module. This list will be empty on many platforms.

C FUNCTIONS

rv2cv_op_cv

Examines an op, which is expected to identify a subroutine at runtime, and attempts to determine at compile time which subroutine it identifies. This is normally used during Perl compilation to determine whether a prototype can be applied to a function call. *cvop* is the op being considered, normally an `rv2cv` op. A pointer to the identified subroutine is returned, if it could be determined statically, and a null pointer is returned if it was not possible to determine statically.

Whether the subroutine is statically identifiable is determined in accordance with the prevailing standards of the Perl version being used. The same criteria are used that the core uses to determine whether to apply a prototype to a subroutine call. From version 5.11.2 onwards, the subroutine can be

determined if the RV that the `rv2cv` is to operate on is provided by a suitable `gv` or `const` op. Prior to 5.11.2, only a `gv` op will do. A `gv` op is suitable if the GV's CV slot is populated. A `const` op is suitable if the constant value must be an RV pointing to a CV. Details of this process may change in future versions of Perl.

If the `rv2cv` op has the `OPpENTERSUB_AMPER` flag set then no attempt is made to identify the subroutine statically: this flag is used to suppress compile-time magic on a subroutine call, forcing it to use default runtime behaviour.

If *flags* has the bit `RV2CVOPCV_MARK_EARLY` set, then the handling of a GV reference is modified. If a GV was examined and its CV slot was found to be empty, then the `gv` op has the `OPpEARLY_CV` flag set. If the op is not optimised away, and the CV slot is later populated with a subroutine having a prototype, that flag eventually triggers the warning "called too early to check prototype".

If *flags* has the bit `RV2CVOPCV_RETURN_NAME_GV` set, then instead of returning a pointer to the subroutine it returns a pointer to the GV giving the most appropriate name for the subroutine in this context. Normally this is just the `CvGV` of the subroutine, but for an anonymous (`CvANON`) subroutine that is referenced through a GV it will be the referencing GV. The resulting `GV*` is cast to `CV*` to be returned. A null pointer is returned as usual if there is no statically-determinable subroutine.

```
CV *rv2cv_op_cv(OP *cvop, U32 flags)
```

cv_get_call_checker

Retrieves the function that will be used to fix up a call to *cv*. Specifically, the function is applied to an `entersub` op tree for a subroutine call, not marked with `&`, where the callee can be identified at compile time as *cv*.

The C-level function pointer is returned in **ckfun_p*, and an SV argument for it is returned in **ckobj_p*. The function is intended to be called in this manner:

```
entersubop = (*ckfun_p)(aTHX_ entersubop, namegv, (*ckobj_p));
```

In this call, *entersubop* is a pointer to the `entersub` op, which may be replaced by the check function, and *namegv* is a GV supplying the name that should be used by the check function to refer to the callee of the `entersub` op if it needs to emit any diagnostics. It is permitted to apply the check function in non-standard situations, such as to a call to a different subroutine or to a method call.

By default, the function is Perl_ck_entersub_args_proto_or_list, and the SV parameter is *cv* itself. This implements standard prototype processing. It can be changed, for a particular subroutine, by "cv_set_call_checker".

```
void cv_get_call_checker(CV *cv, Perl_call_checker *ckfun_p,
        SV **ckobj_p)
```

cv_set_call_checker

Sets the function that will be used to fix up a call to *cv*. Specifically, the function is applied to an `entersub` op tree for a subroutine call, not marked with `&`, where the callee can be identified at compile time as *cv*.

The C-level function pointer is supplied in *ckfun*, and an SV argument for it is supplied in *ckobj*. The function is intended to be called in this manner:

```
entersubop = ckfun(aTHX_ entersubop, namegv, ckobj);
```

In this call, *entersubop* is a pointer to the `entersub` op, which may be replaced by the check function, and *namegv* is a GV supplying the name that should be used by the check function to refer to the callee of the `entersub` op if it needs to emit any diagnostics. It is permitted to apply the check function in non-standard situations, such as to a call to a different subroutine or to a method call.

The current setting for a particular CV can be retrieved by "cv_get_call_checker".

```
void cv_set_call_checker(CV *cv, Perl_call_checker ckfun,
        SV *ckobj)
```

ck_entersub_args_list

Performs the default fixup of the arguments part of an `entersub` op tree. This consists of applying list context to each of the argument ops. This is the standard treatment used on a call marked with `&`, or a method call, or a call through a subroutine reference, or any other call where the callee can't be identified at compile time, or a call where the callee has no prototype.

```
OP *ck_entersub_args_list(OP *entersubop)
```

ck_entersub_args_proto

Performs the fixup of the arguments part of an `entersub` op tree based on a subroutine prototype. This makes various modifications to the argument ops, from applying context up to inserting `refgen` ops, and checking the number and syntactic types of arguments, as directed by the prototype. This is the standard treatment used on a subroutine call, not marked with `&`, where the callee can be identified at compile time and has a prototype.

protosv supplies the subroutine prototype to be applied to the call. It may be a normal defined scalar, of which the string value will be used. Alternatively, for convenience, it may be a subroutine object (a `CV*` that has been cast to `SV*`) which has a prototype. The prototype supplied, in whichever form, does not need to match the actual callee referenced by the op tree.

If the argument ops disagree with the prototype, for example by having an unacceptable number of arguments, a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. In the error message, the callee is referred to by the name defined by the *namegv* parameter.

```
OP *ck_entersub_args_proto(OP *entersubop, GV *namegv,
        SV *protosv)
```

ck_entersub_args_proto_or_list

Performs the fixup of the arguments part of an `entersub` op tree either based on a subroutine prototype or using default list-context processing. This is the standard treatment used on a subroutine call, not marked with `&`, where the callee can be identified at compile time.

protosv supplies the subroutine prototype to be applied to the call, or indicates that there is no prototype. It may be a normal scalar, in which case if it is defined then the string value will be used as a prototype, and if it is undefined then there is no prototype. Alternatively, for convenience, it may be a subroutine object (a `CV*` that has been cast to `SV*`), of which the prototype will be used if it has one. The prototype (or lack thereof) supplied, in whichever form, does not need to match the actual callee referenced by the op tree.

If the argument ops disagree with the prototype, for example by having an unacceptable number of arguments, a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. In the error message, the callee is referred to by the name defined by the *namegv* parameter.

```
OP *ck_entersub_args_proto_or_list(OP *entersubop, GV *namegv,
        SV *protosv)
```

SEE ALSO

B::CallChecker, Devel::CallParser, "cv_set_call_checker" in perlapi

AUTHOR

Andrew Main (Zefram) <zefram@fysh.org>

COPYRIGHT

LICENSE