Polymorphic Types

Document: N3212
Author: Martin Uecker
Date: 2024-01-13

In the past C added various type-generic functions. Unfortunately, those functions are not first-class entities of the language, but macro contraptions that combine preprocessor functionality with compiler magic. Not being first class entities means that they have no proper type on their own and can not be used for higher-order programming, i.e. not passed to other functions to generic higher-level abstractions. An example is:

```
void atomic_store(volatile A *object, C desired);
```

While this is presented in the C standard as a function prototype, this is misleading as this is not actually a function and there is no type in the C type system that could express the type of **atomic_store**. Implementations of such type-generic functions often use compiler magic or non-standard extensions and then require close coupling of the C library and the compiler. Considering that type safety is an important tool to make programming reliable and safe, and that several such interfaces were added to C23, we believe it is time to create a sound foundation for generic functions.

Here we explore the idea of adding a polymorphic type system to C that allows implementation of typesafe polymorphic functions that are truly first-class entities of the language. Those functions should be usable as replacements for traditional **void*** -based generic APIs but add type safety. This then solves several long-standing problems:

1. Traditional void-based APIs can be made typesafe.

2. Implementation of type-generic functions in the standard library is simplified by making them proper (first-class) C functions and the C library is decoupled more from the compiler.

3. Diagnostics for generic functions is improved and the implementation becomes more robust.

4. Type-generic functions can be passed to other functions in higher-order APIs (callbacks, qsort, etc.) to simplify construction of more complicated algorithms. At the same time, code expansion by excessive macro expansion can be avoided.

In principle, the idea developed here is very simple and seamlessly integrates into existing C with three simple additions to the language:

1. A type **_Type** that can store type information:

   ```
   _Type;
   ```

2. An operator **_Typeof** that can produce a compile-time value of type **_Type** from a type name or expression.

```
int i;
_Type T = _Typeof(i);    // produces a value _T"int"
_Type S = _Typeof(int);  // produces a value _T"int"
```

3. A type specifier **_Var(T)** that can be used to construct a type name from a value **T** of type **_Type**. The following for declaration are then all equivalent:

```
int i;
int *ip;
typeof(i) *i;
_Var(_Typeof(i)) *i;
```

Using a run-time value of type **_Type T** in **_Var(T)** can be allowed in general (and is useful), but needs to be restricted to contexts where full type information is not required at compile-time. For example, it is allowed to construct a pointer:

```
_Type T = _Typeof(i);
extern _Var(T) i;             // invalid!
extern _Var(T)* pi = &i;      // ok!
```

In cases where **_Var(T)** is the type of a pointer target it would act just like **void**, so a macro definition that ensures compatibility with old compilers and existing APIs could simply be the following:

```
#define _Var(T) void
```

Nevertheless, an implementation for C2Y would have a keyword **_Var** and support type checking for such types in contexts where T is a compile-time constant according to exact rules.

We can now use such a system to define a first-class type-generic **atomic_store_tg** function (which is similar to what implementations actually have in their run-time libraries):

```
void atomic_store_tg(_Type T,
                     volatile _Atomic(_Var(T)) *object,
                     const _Var(T) *desired);
```

In contrast to the original definition we had to do some changes: Both arguments are passed by pointers. Thus, for purposes of ABI we can simply replace **_Var(T)*** by **void*** and obtain the following first-class C function:

```
void atomic_store_tg(_Type T,
                     volatile _Atomic void *object,
                     const void *desired);
```

If the first argument is a constant of type **_Type** a compiler would then be required by a new constraint to perform static type checking. If it is a run-time value, a run-time type check could optionally be added by a sanitizer. In any case, the type information provides polymorphic type information that is essential for type safe generic API. We intentionally use the exact same syntax for run-time and compile-time types.

If we want to express the original **atomic_store** macro, then this can be done in the following way:

```
#define atomic_store(object, desired)                 \
     atomic_store_tg(_Typeof(desired),                \
                     object,                          \
                     &(_Typeof(desired)){ desired })
```

This macro now adds a primitive form of type inference using **_Typeof** and uses a compound literal to pass the value 'desired' as a pointer. For new interfaces, we believe that making them truly first-cass functions from the beginning such as **atomic_store_tg** is preferable to such macros.

Another example is a type-generic API for sorting. Consider the declaration of **qsort**:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

A generic interface for such a function could be:

```
void sort(_Type T, size_t N, _Var(T) array[N],
          int (*compare)(const _Var(T)*, const _Var(T)*));
```

This then allows run-time type checking that can be performed as part of sanitizers. In the common case where this function is called with a constant type argument **T,** a compiler would be required to perform static type checking:

```
int array[10];
int compare(const int *a, const int *b);
sort(_Typeof(int), 10, array, compare);
```

While it may seem strange to use a regular function argument for static type checking which does not necessarily have to be a compile-time constant, this is not much different to another situation we already have: For arrays - which always depend on an integer size - we have type checking which is also depending on a similar condition: When constant folding leads to an integer constant expression, type checking at compile-time is required. If it does not result in a compile-time constant, we end up with a variably-modified type where then only run-time checking is possible (it is run-time UB, so currently not required but allowed).

```
char x[2];
char (*p)[1 + 1] = &p;   // static type checking, compile-time
int N = 2;
char (*p)[N] = &p;       // variably-modified type, run-time
```

Both cases, the compile-time type checking and the (possible) run-time type checking are unified into a common framework that computes with value expressions which make use of of the same regular C syntax. Note that this unification is a common and powerful feature of dependently typed languages. In a dependently typed language one would typically aim to fully prove type safety at compile-time, but some languages also delegate some type checking to run-time and this seems suitable for low-level languages such as C [1,2].

Note that a generic function would then have to recover the size from the argument of type **\
_Type. Thus, a **sizeof** operation is needed that works also for non-constant T:

```
_Type T = _Typeof(int);
size_t s = _Sizeof(T);
```

Since we already have a run-time **sizeof**, a C2Y compiler would support this with the conventional **sizeof** operator:

```
_Type T = _Typeof(int);
size_t s = sizeof(_Var(T));
```

A definition of the generic sort function in terms of **qsort** can then be:

```
void sort(_Type T, size_t N, _Var(T) array[N],
          int (*compare)(const _Var(T)*, const _Var(T)*))
{
     qsort(array, N, sizeof(Var(T)), compare);
}
```

In summary, the use of these type mechanism allows us to define regular C functions that are type generic but remain first-class functions that can be implemented in a library and passed to other functions. Under specific conditions, we can require compile-time type checking, and run-time checks can be inserted in the more general case.

Implementation

_Type could internally simply be a pointer to a string to a serialized description of the type. Here, we suggest to simply use a character string that contains a C type name:

```
Typeof(int(*)[3]) x;     // -> _T"int(*)[3]"
int i[4];
_Typeof(typeof(i)*) x;   // -> _T"int(*)[4]"
```

This makes it very easy to implement as C compilers know how to print and parse type names. The strings generated in this way are always compile-time constants, so no dynamic memory management is required, and an object of type **_Type** is then simply a pointer that can be passed

around cheaply. There are some minor complications for tagged type: For constructing the representation of such types, the types must be expanded fully in the type description, but only once when there is recursion.

```
_T"struct list { struct list *next; }"
```

The exact rules also need some adaption for nested definition of types with the same tag name, but fundamentally there is no major problem. Of course, other types of string-like serializations could be considered, but defining it to be a valid source representation of a type name would have some appealing properties: Any kind of ABI divergence is avoided and the representation can easily be understood and manipulated by existing tools and humans.

A really simple option might be to even make **_Type** just be identical to strings. Type literals would then correspond to string literals:

```
const char type_string[] = _Typeof(int);
_Var("int") i;
```

Possibly, we could include additional information into **_Type**, such as the size of the type:

```
typedef struct _type {
     const char* descr;  // pointer to compile-time string constant
     size_t size;        // size of the type
}* _Type;
```

A simplified approximation could be implemented like this already today:

```
struct _type {
     size_t size;
     const char* descr;
}* _Type;

#define _Typeof(T) &(static struct _type){ sizeof(T), # T }
```

In fact, a somewhat restricted and non-checking implementation of this proposal can be implemented in a way which is fully backwards compatible with older compilers:

https://godbolt.org/z/YscGb511r

With a more sophisticated builtin version to fully support constructing description for all types, a more realistic implementation could look like:

```
#define _Typeof(T) &(static struct _type){ \
          __builtin_type_string(T), sizeof(T) }
```

Here, the new builtin **__builtin_type_string(T)** produces a string literal for the type name of C. A compiler with full support would then also need to support type-checking of types including **_Var(T)** under certain conditions where T is a compile-time constant simply by invoking its existing type checking machinery.

Void Pointer Compatibility

We want to point out one complication with this scheme: While all pointer types to objects can be converted to a void pointer and back, the representation of different pointer types might be different and incompatible. While this seems not actually the case on most implementations, it is not currently guaranteed that they are the same (and this may affect embedded platforms).

Consider the following example:

```
void foo(_Type T, const _Var(T)* v, void (*p)(_Var(T)* s))
{
     (*p)(v);
}

void my_int_cb(int *ip);

foo(_Typeof(int), &(int){ 3 }, my_int_cb);
```

This requires that

```
void (*p)(void *);
```

and

```
void (*p)(int *);
```

are either compatible or implicitly convertible types. For functions referred to directly, a wrapper could be created by the compiler:

```
void my_int_cb(int *ip);

void my_int_stub(void *p) { my_int_cb(p); };
foo(_Typeof(int), &(int){ 3 }, my_int_stub);
```

Unfortunately, this has two problems: Firstly, it breaks comparison of pointers when done implicitly, because wrapper created at different points for the same function would not compare equally. Secondly, general function pointer conversion would require the creation of some kind of stub function or trampoline that captures the original pointer to be wrapped.

A simpler solution is to make **_Var(T)** be compatible to a void pointer as described above with the composite type being **_Var(T)**. Thus, pointers derived from **_Var(T)** are required to have the same representation as a void pointer, but are not required to have the same representation as a native pointer to the type denoted by T.

Variably-Modified Types

For variably modified types the type depends on a run-time value. While **typeof** evaluates its argument when it has a variably modified type, we suggest that **_Typeof** should never evaluate its

argument and return a constant value that does not contain information about the run-time value the type depends on:

```
int n = 3;
int i[n];
_Type T = _Typeof(i); // _T"int[*]"
```

An astute reader may have noticed that the type **_Var(T)** can also be variably-modified, i.e. when T is not a constant. This design is intentional as it makes it possible to treat generic functions as full first-class citizens that can be compiled into library functions (cf. the generic sort function above).

Extensions and Applications

There are many possible extensions and applications of this scheme. We discuss some basic ideas:

Type Inference

We deliberately did not include any kind of automatic type inference in the proposal. It should be noted that in some cases type inference would make it possible to make existing functions safe where a **_Type** argument can not retrospectively be added. For example, the declaration

```
void foo(void *a, const void *b, size_t size);
```

could be retrospectively changed to

```
void foo(_Var(T) *a, const void *_Var(T), size_t size);
```

while preserving the ABI as pointed out in N2853.

In general, it would be desirable to have a declaration for T to avoid name lookup ambiguities (e.g. the declaration is nested in other declarations), which could potentially be achieved with a parameter forward declaration (N3207) that is then not followed by an actual parameter:

```
void foo(_Type T; _Var(T) *a, const void *_Var(T), size_t size);
```

A compiler could derive the value of T by inferring a type from the other arguments. Alternatively, such functions declarations could be replaced by a macro that does the type inference explicitly (as shown above for the **atomic_store** generic function). For certain function suppressing the macro definition could resolve to a conventional definition using void pointers, preserving the first-class nature of such functions.

Another scenario where type inference might be useful is when assigning a pointer to a generic function to a more specific pointer type, which is similar to the automatic specialization proposed for type-generic lambdas (N2890). The framework developed here implies that this could be considered as generic problem of function specialization, where a compiler derives the required values for arguments of type **_Type** and also **size_t** automatically. We do not discuss such ideas further because we believe that type inference is a more complicated topic.

Variadic Arguments

Another potential useful application is to pass type information to variadic functions. One often finds code such as the following in C projects:

```
enum my_type { MY_INT, MY_DOUBLE };
int generic_print(enum my_type ty, ...)
{
     va_list ap;
     va_start(ap);
     if (ty == MY_INT)
          printf("%d\n", std_arg(ap, int));
     else if (ty == MY_DOUBLE)
          printf("%d\n", std_arg(ap, double));
     va_end(ap);
}
```

Here the programmer encodes information about types into enumerators or even more complex data structures. Such code could become simpler with a builtin type **_Type**. For example, with help of some type comparison function, this could be rewritten to:

```
int generic_print(_Type T, ...)
{
     va_list ap;
     va_start(ap);
     if (type_cmp(T, _Typeof(int))
          printf("%d\n", std_arg(ap, int));
     else if(type_cmp(T, _Typeof(int, double))
          printf("%f\n", std_arg(ap, double));
     va_end(ap);
}
```

Dynamic Functions Calls

A builtin type **_Type** is useful for dynamic construction of function calls. For example, libffi has its own infrastructure for describing types and function calls, which could be replaced by **_Type.**

```
void ffi_call(_Type ret, int N, const _Type args[N],
              _Var(ret) *retval, void (*fun)(), ...);
```

References

1. C. Flanagan. *Hybrid Type Checking.* POPL'06. 2006;245–256. https://doi.org/10.1145/1111037.1111059

2. J. Condit et al. *Dependent types for low-level programming.* ESOP'07. 2007;520–535.

Acknowledgment: Alex Celeste for insightful discussions.