```
Proposal for C23
WG14 2853

Title:                The `void`-_which-binds_: typesafe parametric polymorphism
Author, affiliation: Alex Gilding, Perforce
Date:                 2021-10-08
Proposal category:    New features
Target audience:      Compiler/tooling developers, library developers,
application developers
```

Abstract

C has a built-in mechanism for handling generically-typed data: `void *`. Unfortunately, this has a problem: it achieves genericity by simply discarding all type information!

We propose a mechanism to define native functions that are *both* polymorphic *and* strongly-typed, as commonly seen in functional programming languages or those that have adopted functional idioms. The mechanism does not impose any runtime or ABI burden.

The `void`-*which-binds*: type-safe parametric polymorphism

Reply-to: Alex Gilding (agilding@perforce.com)
Document No: N2853
Revises Document No: N/A
Date: 2021-10-08

Summary of Changes

N2853

- original proposal

Introduction

C can define polymorphic functions by using `void *`, but this comes at the cost of complete type erasure.

This means that while a function can be defined to operate on any type of data, it trades the ability to do so for the lost ability to statically check that the operations it wants to perform on that data are actually valid for its concrete type. It cannot guarantee that two operands to be compared originate from the same kind of data, and it cannot even guarantee that the comparison operator intends to work with them if they are!

In many other languages, expressing the necessary constraints is easy. In many languages influenced by ML, "type variables" can be used in the function signature; while in C++, templates can be used to parameterize a function over types (although templates are not parametric in quite the same way).

We propose a matched pair of two attributes which can appertain to a pointed-to `void`, which force the pointer to either bind to a concrete type (for operator implementations), or to bind *consistently* with other `void` pointers within the same top-level function signature. Because these are attributes, they impose no runtime cost; and if they are removed from a correct program, ensure it will remain binary-identical to the type-safe version.

Rationale

Other languages, especially those descending from or influenced by ML, have the notion of "type variables", by which function (and other) types can be parameterized: at the point of use, an operand type is substituted into the type of the function, and if the same type variable appears in two places in the function type, it has to have the same binding to a type deduced from the operands, enforcing type safety.

So in an ML-like or Haskell-like language, a function can have a type like this:

```
map :: forall a. ([a], (a -> a)) -> [a]
```

...which we can understand to take some sequence of elements of type `a`, a callback which transforms objects of type `a` to other objects also of type `a`, and returns another sequence which (we will assume) contains the results of each transformation. We can't call this function with a callback that operates on objects of a type not compatible with the objects in the input buffer.

Informatively, a similar C++ signature might be:

```
template <typename A>
auto map (Seq <A> const &, Function <A (A)> const &) -> Seq <A>;
```

Though it is important to remember that C++ templates are not functions; they merely instantiate into *monomorphic* functions, so this is not the same. Polymorphic functions in Haskell and ML have a truly polymorphic type for a single identity; the same function can be referenced, passed around, and still used polymorphically at a distance. This property is also true of C functions that use `void *` - they can be referenced by pointer and still applied to different argument types - and is one we wish to preserve.

To express this idiomatically in C, we'd use `void *`, and the function signature would probably look something like:

```
void map (void * in, void * out, void (* op) (void const *, void *), void * (*
step) (void *));
```

Nothing prevents us from providing a mistyped `op` - or even a mistyped `step`: not even navigating a polymorphic sequence is assured to make sense!

Surprisingly, this is actually possible to enforce with a lot of supporting verbosity in pure C11. However, the technique is too verbose and fragile for real-world use and does not scale at all. Examples are provided at the end of an untyped program which can miscompile, a typechecked C11 program, and an equivalent typechecked program using the `void`-*which-binds*.

Proposal

We propose to add two new attributes which can appertain to the type `void`, when it appears as the *referenced type* of a pointer type used as either a function parameter or a function return. These are:

```
[[bind_var (id)]]
[[bind_type (type)]]
```

where `id` is an identifier that introduces a *type variable*, and `type` is a concrete type name other than `void` (it does not otherwise need to be complete).

The names of these attributes are tentative.

Syntactically, because these appertain to the type `void` itself, they appear immediately after it in any given pointer type specifier. They may also appertain to a typedef of `void`.

Both of these attributes add *constraints* which restrict which pointer types the given parameter or return value may be converted to without an explicit cast.

Effectively both of these attributes add virtual qualifiers that associate with the pointed-to type and therefore cannot be discarded except by explicit cast-off.

`bind_type`

The simpler attribute is `bind_type`. This adds the *constraints* that the pointer with the annotated referenced type can only implicitly convert to or from a (suitably qualified) pointer to `type`, or be assigned to another (suitably qualified) pointer to `void` with the same annotation.

For example:

```
void [[bind_type (float)]] const *
float_op_impl (void [[bind_type (float)]] const * p1, void [[bind_type (float)]]
* p2)
{
  float const * fp1 = p1; // OK, bound type
  float * fp2 = p2; // OK

  int  const * ip1 = p1; // new constraint violation
  void const * vp1 = p1; // new constraint violation
  int  const * ip2 = p2; // new constraint violation
  void const * vp2 = p2; // new constraint violation

  fp2 = p1; // (existing constraint violation)

  ip2 = (int  *)p2; // OK, explicit cast
  vp2 = (void *)p2; // OK, explicit cast-off of bind_type

  return fp1; // OK, bound type
  return vp1; // new constraint violation
  return (float const *)ip1; // ...OK at least by these rules

  return p1; // OK, same bind_type
}
```

This attribute is not intended to be used in polymorphic functions themselves, but rather in the operators passed to the polymorphic function, which are monomorphic in themselves but usually will require a signature that explicitly works with `void *` (because `int * (*) (int *)` and `void * (*) (void *)` are not convertible or guaranteed to have similar representations), in order to be passed to the actual polymorphic function.

`bind_var`

This attribute is used on the parameters and return type of the actual polymorphic function. It is also appertained to the parameters and return type of any callback operators accepted by the polymorphic function, within its own signature.

`bind_var` adds the *constraint* that whatever the referenced types of the pointers that are converted to or from `void *` in the argument types and returned value conversion, must be consistent between all `void` pointers annotated with the same `id`, within each instance of a call to that function.

In other words it introduces a *type variable* and all `void *` annotated with the same *type variable* must convert to or from the same pointer type.

For example:

```
void [[bind_var (A)]] *
select_first (void [[bind_var (A)]] * a, void [[bind_var (A)]] * b, void
[[bind_var (B)]] * c)
```

```
{
  return a; // OK, a must have bound to an A*
  return b; // OK, b must have bound to an A*
  return c; // new constraint violation, c may have bound to a different type
}

int * ip;
float * fp;

ip = select_first (ip, ip, fp); // OK, a and b are both A
fp = select_first (fp, fp, ip); // same

ip = select_first (ip, ip, ip); // OK, A and B can both be the same, it just
doesn't know that

ip = select_first (fp, fp, ip); // new constraint violation, mismatched A
between return/param
select_first (ip, fp, fp);      // new constraint violation, mismatched A
between params 1 & 2
```

The two attributes combine: `bind_var` can bind its *type variable* to the type named in a `bind_type` annotation, to allow operator callbacks to be type-checked against data arguments to a higher-order polymorphic function:

```
void map (
    void [[bind_var (A)]] * in
  , void [[bind_var (A)]] * out
  , void (* op) (void [[bind_var (A)]] const *, void [[bind_var (A)]] *)
  , void [[bind_var (A)]] * (* step) (void [[bind_var (A)]] *)
);
```

In this declaration of a polymorphic map (implementation is in the Examples section), all of the `void *` type components have been annotated as having to bind to the same concrete object type. Therefore:

```
int i1[10], i2[10];
float f1[10], f2[10];

void int_op (void [[bind_type (int)]] const *, void [[bind_type (int)]] *);
void [[bind_type (int)]] int_step (void [[bind_type (int)]] *);

void float_op (void [[bind_type (float)]] const *, void [[bind_type (float)]]
*);
void [[bind_type (float)]] float_step (void [[bind_type (float)]] *);

map (i1, i2, int_op, int_step); // OK, all void * bind to int *
map (f1, f2, float_op, float_step); // OK, all void * bind to float *

map (i1, i2, float_op, float_step); // new constraint violation - operator types
do not match data
map (i1, f2, int_op, float_step); // new constraint violation - operator and
data types mismatch
```

Finally, if a polymorphic function is called from within another polymorphic function, the *type variables* bind to the *type variables* of the `void *` pointers passed directly to it, rather than to `void`:

```
// yikes
void map2 (
    void [[bind_var (A)]] * in1
```

```
  , void [[bind_var (A)]] * out1
  , void (* op1) (void [[bind_var (A)]] const *, void [[bind_var (A)]] *)
  , void [[bind_var (A)]] * (* step1) (void [[bind_var (A)]] *)

  , void [[bind_var (B)]] * in2
  , void [[bind_var (A)]] * out2
  , void (* op2) (void [[bind_var (B)]] const *, void [[bind_var (B)]] *)
  , void [[bind_var (B)]] * (* step2) (void [[bind_var (B)]] *)
) {
  map (in1, out1, op1, step1); // OK, all void * bind to A *
  map (in2, out2, op2, step2); // OK, all void * bind to B *

  map (in1, out1, op2, step2); // new constraint violation - operator types do
not match data
  map (in1, out2, op1, step2); // new constraint violation - operator and data
types mismatch
}
```

We would propose that the `bind_var` attribute apply immediately to the functions `qsort` and `bsearch` in `stdlib.h`.

Alternatives

The names and exact placement of the attributes can be subject to further discussion. In practice we expect that users would wrap them inside macros anyway, for instance:

```
#define Void(T) void [[bind_var (T)]]
#define Bind(T) void [[bind_type (T)]]

void map (
    Void (A) * in
  , Void (A) * out
  , void (* op) (Void (A) const *, Void (A) *)
  , Void (A) * (* step) (Void (A) *)
);

void int_op (Bind (int) const *, Bind (int) *);
```

Overloads and `_Generic` do not provide a suitable alternative to this feature. The ad-hoc polymorphism provided by `_Generic` operates on a fixed set of known types. The operator name that it can be used to construct is a second-class language feature that does not have a single address. To pass such an operator around, it must be deconstructed somehow and a non-generic overload selected.

This functionality is completely orthogonal; where an ad-hoc polymorphic "overloaded" function provides *different* suitable implementations for a number of different types, a parametrically polymorphic function is *completely unaware* of the concrete type of its arguments and always provides the exact same implementation regardless of it.

Templates are similarly orthogonal: they do not create a first-class polymorphic language feature (remaining polymorphic when passed by value), but instantiate into separate monomorphic functions. They can also specialize, which betrays the principle of always providing the *exact* same body implementation down to the instruction level.

Impact

The implementation impact is lower than it seems.

Firstly, there is **no ABI impact** whatsoever from this feature. Because the constraints are applied by attributes, they do not change the representation type of the pointers to generic data, which remain `void *` at all times. There is no binary difference between a program that chooses to make use of this typechecking feature and one that does not.

This is different from C++, where a template function instantiates into multiple implementations with semantically separate identities and which all have different signature types. There remains only one polymorphic function, with one type, and whose type remains polymorphic even after it is referenced indirectly.

This follows directly from the principle that a correct program with standard attributes *remains* correct and has the exact same observable meaning if the attributes are removed.

Secondly, if polymorphic type checking proves too difficult for a smaller implementation, since these constraints are onyl introduced by attribute - the implementation can choose to be unable to diagnose violations of the constraints and will still be conforming, because it will always compile a correct program to exactly the same meaning as a higher-end compiler that does understand the attributes. Therefore, implementations are eased-in to needing to support polymorphism, and do not need to provide full checking in order to conform.

It is possible and entirely plausible that only analysis tools and not primary compilers would ever bother to implement the constraints for these attributes.

Thirdly, implementation experience with Helix QAC suggests that ML-style type inference with unification is "not too difficult" for a small team (one) to implement in a short amount of time. We found that C could easily be extended with backpropagating inferred pointer types, which are used to drive a number of different type-based (proprietary) analyses. (In QAC these properties are mostly inferred from usage rather than specified explicitly by user attributes.)

ML subset compilers are frequently implemented as student projects, so this is a widely understood feature in the broader compiler community.

Proposed wording

Changes are proposed against the wording in C2x draft n2596. Bolded text is new text.

Modify 6.7.11 "Attributes":

Modify 6.7.11.1 to add the two new standard attribute names:

> The identifier in a standard attribute shall be one of:
> **`bind_type`**
> **`bind_var`**
> `deprecated`
> `fallthrough`
> `maybe_unused`
> `nodiscard`

Note that 6.7.11.1 paragraph 3 and note 166 **remain true** for the new attributes without changes.

Add two new sections after 6.7.11.5:

6.7.11.6 The `bind_type` attribute

Constraint

The `bind_type` attribute shall be applied to the `void` referenced type of a pointer in the return or parameter types of a function declaration or function pointer, or to the `void` referenced type of the pointer type of a local variable within a function that uses `bind_type` in its signature.

`bind_type` requires an argument clause, which shall have the form `(` *type-name* `)`

Semantics

The `bind_type` attribute indicates that while a pointer has the concrete type of a pointer to `void`, it is constrained to only allow implicit conversions to or from a suitably-qualified pointer to `type`, where `type` is specified by the *type-name*. Conversions to any other pointer types without an explicit cast are prohibited.

The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value 202111L when given `bind_type` as the *pp-tokens* operand.

Recommended Practice

Implementations should use `bind_type` on the parameters and return values of callback functions that internally operate on a single type, but use pointers to `void` in their signature in order to be callable from a generic higher-order function.

This attribute allows the usage of the callback to be type-checked against its operating type in addition to the forced signature type.

EXAMPLE

This callback is intended to be passed to `qsort` to sort an array of `float` values. Its operating parameter types are float pointers, but the signature of `qsort` requires them to be declared as pointers to `void`:

```
int float_compare (const void [[bind_type (float)]] * l
                 , const void [[bind_type (float)]] * r) {
    const float * fl = l;
    const float * fr = r;

    return *fl < *fr ? -1
         : *fl > *fr ? +1
         :             0;
}
```

6.7.11.7 The `bind_var` attribute

Constraint

The `bind_var` attribute shall be applied to the `void` referenced type of a pointer in the return or parameter types of a function declaration or function pointer, or to the `void` referenced type of the pointer type of a local variable within a function that uses `bind_var` in its signature.

`bind_var` requires an argument clause, which shall have the form `(` *identifier* `)`

Semantics

The `bind_var` attribute indicates that while a pointer has the concrete type of a pointer to `void`, it is constrained to only allow assignment within a function body to or from a suitably-qualified pointer to `void` that has been annotated with the same `ident`, where `ident` is specified by the *identifier*.

Conversions to any other pointer types, including a pointer to un-attributed `void`, without an explicit cast are prohibited within the body of the function that introduces the binding.

Implicit conversion from another object pointer type in the argument list, or to another object pointer type in the return, are permitted at the function's call site. In all cases where two attributed pointers to `void` in the function signature have the same identifier, they shall only convert to or from the same pointer to object type footnote).

The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value 202111L when given `bind_var` as the *pp-tokens* operand.

footnote) there is no restriction against two attributed pointers to void with different `ident`s converting to or from the same type outside the function, but internally the function shall treat them as unrelated.

Recommended Practice

Implementations should use `bind_var` in the parameters and return values of a polymorphic function to indicate that two separate generic pointers in its signature are intended to work on the same concrete type for any given invocation.

This attribute allows usage of the polymorphic function to be type-checked against the concrete types of its arguments and destination, as well as the expected types to be operated on by any callback operators passed to it.

EXAMPLE

This polymorphic function returns a reference to an element within the first buffer, It may not return a reference to an element within the second buffer.

```
int i1[10];
float f1[10];

void [[bind_var (A)]] * first (void [[bind_var (A)]] * p1
                               , void [[bind_var (B)]] * p2) {
    return p1; // correct
    // return p2; // constraint violation
```

```
}

int   * ip = first (i1, f1); // OK
float * fp = first (f1, i1); // OK
ip = first (i1, i1); // also OK, the function just doesn't know A and
B are compatible here
fp = first (i1, f1); // constraint violation, A and A do not match
```

`qsort` is a polymorphic function that needs to accept a compare function that will work on its working array, so the following signature communicates this:

```
void my_qsort (void [[bind_var (T)]] * base
            , size_t nmemb
            , size_t size
            , int (*compar)(const void [[bind_var (T)]] *
                         , const void [[bind_var (T)]] *));)

my_qsort (f1, 10, sizeof (float), float_compare); // OK
my_qsort (i1, 10, sizeof (int), float_compare); // constraint
violation, A and A do not match
```

Modify 7.22.5.1 "The `bsearch` function", paragraph 1, to include `bind_var` in the signature:

```
#include <stdlib.h>
void * bsearch (const void [[bind_var (T)]] *key
             , const void [[bind_var (T)]] *base
             , size_t nmemb
             , size_t size
             , int (*compar)(const void [[bind_var (T)]] *
                          , const void [[bind_var (T)]] *));
```

Modify 7.22.5.2 "The `qsort` function", paragraph 1, to include `bind_var` in the signature:

```
#include <stdlib.h>
void qsort (void [[bind_var (T)]] *base
          , size_t nmemb
          , size_t size
          , int (*compar)(const void [[bind_var (T)]] *
                       , const void [[bind_var (T)]] *));
```

Modifications to the text of 7.22.5 are not required.

References

- C23 n2596
- Parametric polymorphism
- Towards type-checked polymorphism

Examples

Example 1: untyped

```
// basic array mapper
// we can use it with mis-typed arguments

typedef void (* Mutate) (void *, void *);
typedef void * (* Step) (void *);

void addOneInt   (void * in, void * out) {   *(int *)out =   *(int *)in +
1;    }
void addOneFloat (void * in, void * out) { *(float *)out = *(float *)in +
1.0f; }

void * step_float (void * p) { return (float *)p + 1; }
void * step_int   (void * p) { return   (int *)p + 1; }

int ia[10];
float fa[10];

void map (void * array_in, void * array_out, int size, Mutate mut, Step step) {
    void * in = array_in;
    void * out = array_out;
    for (int i = 0; i < size; ++ i, in = step (in), out = step (out)) {
        mut (in, out);
    }
}

void incrArrays (void) {
  map (ia, ia, 10, addOneInt, step_int);
  map (fa, fa, 10, addOneFloat, step_float);

  // oh no: this also compiles, because of void*
  map (fa, fa, 10, addOneInt, step_float);
  map (ia, fa, 10, addOneInt, step_float);
  map (ia, fa, 10, addOneInt, step_float);
  map (ia, ia, 10, addOneInt, step_float);
  map (fa, fa, 10, addOneInt, step_int);

  map (ia, ia, 10, addOneFloat, step_int);
  map (ia, fa, 10, addOneFloat, step_int);
  map (ia, ia, 10, addOneFloat, step_int);
  map (ia, ia, 10, addOneFloat, step_int);
  map (ia, ia, 10, addOneFloat, step_float);
}
```

Example 2: semiautomatic typechecking with brittle verbosity (C11)

```
// basic array mapper
// enhanced with type checking despite accepting arrays of any type - checks
// that the operand kind of the mapped function matches the array element type
//  i.e. map :: ([T], T -> T) -> [T]

typedef void (* Mutate) (void *, void *);
typedef void * (* Step) (void *);

void addOneInt   (void * in, void * out) {   *(int *)out =   *(int *)in +
1;    }
void addOneFloat (void * in, void * out) { *(float *)out = *(float *)in +
1.0f; }

void * step_float (void * p) { return (float *)p + 1; }
void * step_int   (void * p) { return   (int *)p + 1; }

int ia[10];
float fa[10];

void map_impl (void * array_in, void * array_out, int size, Mutate mut, Step
step) {
  void * in = array_in;
  void * out = array_out;
  for (int i = 0; i < size; ++ i, in = step (in), out = step (out)) {
    mut (in, out);
  }
}


#define FunctionDescriptor(Type, Func) union { Func func; Type T; }

#define same_type(A, B) _Generic(1 ? (A) : (B)  \
  , void *: 0                                   \
  , void const *: 0                             \
  , void volatile *: 0                          \
  , void const volatile *: 0                    \
  , default: 1)
#define check_same_type(A, B) _Static_assert (same_type (A, B), "types must
match");

#define check_array_size

#define map(in, out, size, mut, step) do {                \
  check_same_type ((in), (out));                          \
                                                          \
  check_same_type ((in), &(mut).T);                       \
  check_same_type ((in), &(step).T);                      \
                                                          \
  map_impl ((in), (out), (size), (mut).func, (step).func); \
} while (0)


typedef FunctionDescriptor (int, Mutate) MutInt;
typedef FunctionDescriptor (int, Step) StepInt;
typedef FunctionDescriptor (float, Mutate) MutFloat;
typedef FunctionDescriptor (float, Step) StepFloat;

MutInt addOneInt_g;
StepInt stepInt_g;

MutFloat addOneFloat_g;
```

```
StepFloat stepFloat_g;


void incrArrays (void) {
  map (ia, ia, 10, addOneInt_g, stepInt_g);
  map (fa, fa, 10, addOneFloat_g, stepFloat_g);

  // no longer compile!
  // map (fa, fa, 10, addOneInt, step_float);
  // map (ia, fa, 10, addOneInt, step_float);
  // map (ia, fa, 10, addOneInt, step_float);
  // map (ia, ia, 10, addOneInt, step_float);
  // map (fa, fa, 10, addOneInt, step_int);

  // map (ia, ia, 10, addOneFloat, step_int);
  // map (ia, fa, 10, addOneFloat, step_int);
  // map (ia, ia, 10, addOneFloat, step_int);
  // map (ia, ia, 10, addOneFloat, step_int);
  // map (ia, ia, 10, addOneFloat, step_float);
}
```

Example 3: polymorphic typechecking with `void`-*which-binds*

```
// basic array mapper
// enhanced with type checking despite accepting arrays of any type - checks
// that the operand kind of the mapped function matches the array element type
//  i.e. map :: ([T], T -> T) -> [T]

#define Auto(T) void [[bind_var (T)]]
#define Bind(T) void [[bind_type (T)]]

#define Mutate(T, N) void (* N) (Auto (T) *, Auto (T) *)
#define Step(T, N) Auto (T) * (* N) (Auto (T) *)

void addOneInt   (Bind (int)   * in, Bind (int)   * out) {   *(int *)out =
*(int *)in + 1;    }
void addOneFloat (Bind (float) * in, Bind (float) * out) { *(float *)out =
*(float *)in + 1.0f; }

Bind (float) * step_float (Bind (float) * p) { return (float *)p + 1; }
Bind (int)   * step_int   (Bind (int)   * p) { return   (int *)p + 1; }

int ia[10];
float fa[10];

void map (Auto (A) * array_in, Auto (A) * array_out, int size, Mutate(A, mut),
Step (A, step)) {
    Auto (A) * in = array_in;
    Auto (A) * out = array_out;
    for (int i = 0; i < size; ++ i, in = step (in), out = step (out)) {
        mut (in, out);
    }
}

void incrArrays (void) {
  map (ia, ia, 10, addOneInt, step_int);
  map (fa, fa, 10, addOneFloat, step_float);

  // no longer compile!
  // map (fa, fa, 10, addOneInt, step_float);
  // map (ia, fa, 10, addOneInt, step_float);
  // map (ia, fa, 10, addOneInt, step_float);
  // map (ia, ia, 10, addOneInt, step_float);
  // map (fa, fa, 10, addOneInt, step_int);

  // map (ia, ia, 10, addOneFloat, step_int);
  // map (ia, fa, 10, addOneFloat, step_int);
  // map (ia, ia, 10, addOneFloat, step_int);
  // map (ia, ia, 10, addOneFloat, step_int);
  // map (ia, ia, 10, addOneFloat, step_float);
}
```