Enabling Generic Functions and Parametric Types in C (WG14 N2698)

Why

- Macros are unanalyzable and fragile.
- void* is non-uniform and ineffective.

And _Generic is just a type-specific dispatcher.

Design

- Parametric types
- Generic functions
- Constraint-based type inference and unification
- Monomorphization

Design goal

- Parametric types
- Generic functions
- Constraint-based type inference and unification
- Monomorphization

1. Compiler pluggability and C compatibility
2. Expressiveness

Parametric types and generic functions

- A type that is specified with the _Any parametric type specifier (allowed in certain contexts) is a parametric type; a function declared with a parameter of parametric type is a generic function.

```
void f(_Any(t) p) {}
void g(_Any(t) p, int q) {}
void h(int* p, _Any(t) q, double r) {}
```

- There are also (possibly qualified) parametric array, pointer, and function types.

```
void f(_Any(t) p) {}
void g(_Any(t) p[]) {}
void h(_Any(t)* p) {}
void i(int (*p) (_Any(t))) {}
```

Parametric types in the return and body of functions

- A parametric type may only appear in the return or body of a function if it also appears in a parameter of said function.

```
_Any(t) identity(_Any(t) p) { return p; }
_Any(t) f(_Any(t)* p) { return *p; }
_Any(t)* g(_Any(t)* p) { return 0; }
```

```
void swap(_Any(t)* p, _Any(t)* q)
{
    _Any(t) v;
    v = *p;
    *p = *q;
    *q = v;
}
```

Parametric types in a struct

- A parametric type may appear as the type of a struct field.
  - Its identifier is the underscore, _.
  - It may not be specified as a pointer, array, or function type.
  - It may not be specified with a qualifier.

  (See 2.3.4 for the justifications around the above stipulations.)

```
struct a { _Any(_) m; };
struct b { _Any(_) m; _Any(_) n; };
struct c { _Any(_) m; _Any(_) n; const int o; };
struct d { _Any(_) m; struct d* n; };
```

Instantiation request of parametric structure types

- The instantiation of parametric structure types is requested explicitly.
- Such a request may be an immediate or pending one, but it's always fulfilled.
  - A request is pending if it's inside a generic function.

```
struct a { _Any(_) m; };
struct b { _Any(_) m; _Any(_) n; };
typedef struct a c;
void f(_Any(t) p)
{
    struct a _With(m:_Any(t)) x;
    struct b _With(m:int, n:double) y;
}
void _()
{
    struct a _With(m:int) x;
    c _With(m:int const*) y;
    struct b _With(m:double, n:double)* z;
    const struct b _With(m:double, n:struct a _With(m:int)) w;
}
```

Instantiation request of parametric (bound) function types

- The instantiation of parametric function types is requested implicitly at instantiation expressions.
- Such a request:
  - Is transitive, encompassing a possible chain of other instantiation expressions.
  - May not be fulfilled:
    - If and only if, for each pair of instantiation expression and generic function in the request, their typing constraint is satisfiable, will the request be fulfilled.

Instantiation request of parametric (bound) function types

```
struct a { int m; };
struct b { _Any(_) m; };
void f(_Any(t) p) {}
void g(_Any(t)* p) { f(*p); }
void h(_Any(t)* p) { const _Any(t) x; p = &x; }
_Any(t) i(_Any(t) p) { return p + 1.0; }
void j(_Any(t)* p, _Any(u) q) { *p = q; }
int k(_Any(t) p) { return p->m; }
double l(struct b _With(m:_Any(t))* p) { return p->m; }
_Any(t) identity(_Any(t) p) { return p; }
void _()
{
    int x;
    f(x);
    f(&x);
    (void (*)(double))f;
    g(&x);
    int* y;
    g(y);
    h(y);
    i(1);
    j(y, 1);
    j(0, 1.0);
    struct a* z;
    k(z);
    struct b _With(m:double)* w;
    l(w);
    identity(identity)(1);
}
```

Fulfilled

```
struct a { int m; };
struct b { int* n; };
struct c { _Any(_) m; };
void f(_Any(t)* p) {}
void g(_Any(t) p) { p + 1.0; }
void h(_Any(t)* p) { _Any(t) x; *p = x; }
_Any(t) i(_Any(t) p) { int i = p; return p + 1.0; }
void j(_Any(t)* p, _Any(u)* q) { p = q; }
int k(_Any(t) p) { return p->n; }
int l(struct b _With(m:_Any(t))* p) { return p->m; }
void _()
{
    int* x;
    f(*x);
    (void(*)(double))f;
    g(x);
    const int* y;
    h(y);
    i(*x);
    struct a* z;
    j(z, x);
    k(z);
    struct b* w;
    k(w);
    struct c _With(m:double)* v;
    l(v);
}
```

Unfulfilled

(See 2.3.2 for the explanations.)

Parametric types instantiation

1. Structure Types Instantiation - 1st Iteration
2. Function Types Instantiation
3. Structure Types Instantiation - 2nd Iteration

Parametric types instantiation

1. Structure Types Instantiation - 1st Iteration

   For immediate requests.

   a. Synthesizes a structure specialization of the parametric structure type.
   b. Patches the type's specifier with a reference to the synthesized structure specialization.

Parametric types instantiation

1. Structure Types Instantiation - 1st Iteration
2. Function Types Instantiation

   If the typing constraint of instantiation expressions and generics functions is satisfiable.

   a. Synthesizes a function specialization of the generic function.
   b. Patches the expression with a reference to the synthesized function specialization.
   c. Repeats (a) and (b) for the chained expressions.

Parametric types instantiation

1. Structure Types Instantiation - 1st Iteration
2. Function Types Instantiation
3. Structure Types Instantiation - 2nd Iteration

   The same as 1 but for pending requests.

Parametric types instantiation - Examples

- Initial program

```
struct a { _Any(_) m; };
struct b { double* n; };
_Any(t)* f(_Any(t) p) { struct a _With(m:_Any(t)*)* x; return x->m; }
int g(_Any(t)* p) { f(p->m); return 1; }
int h(double* (*p) (_Any(t))) { return h(p); }
double* i(struct b* p) { return p->n; }
void _()
{
    double x;
    f(x);
    struct a _With(m:int) y;
    g(&y);
    h(i);
}
```

Parametric types instantiation - Examples

- After Stage 1

```
struct a { _Any(_) m; };
[[@spec_struct !id:1 !tag:a !subs:tyof(m)->int]]
struct b { double* n; };
_Any(t)* f(_Any(t) p) { struct a _With(m:_Any(t)*)* x; return x->m; }
int g(_Any(t)* p) { f(p->m); return 1; }
int h(double* (*p) (_Any(t))) { return h(p); }
double* i(struct b* p) { return p->n; }
void _()
{
    double x;
    f(x);
    [[@spec !id:1]] y;
    g(&y);
    h(i);
}
```

Parametric types instantiation - Examples

- After Stage 2, assuming:
  - All typing constraints are satisfiable.
  - An algorithm of constraint-based type inference via unification that extends that of *Type Inference for C: Applications to the Static Analysis of Incomplete Program* (https://dl.acm.org/doi/fullHtml/10.1145/3421472)

```
struct a { _Any(_) m; };
[[@spec_struct !id:1 !tag:a !subs:tyof(m)->int]]
struct b { double* n; };
_Any(t)* f(_Any(t) p) { struct a _With(m:_Any(t)*)* x; return x->m; }
[[@spec_func !id:2 !name:f !subs:t->double]]
[[@spec_func !id:3 !name:f !subs:t->int]]
int g(_Any(t)* p) { f(p->m); return 1; }
[[@spec_func !id:4 !name:g !subs:t->[[@spec !id:1]] !chains:[[@spec !id:3]]]]
int h(double* (*p) (_Any(t))) { return h(p); }
[[@spec_func !id:5 !name:h !subs:t->struct b* !chains:[[@spec !id:5]]]]
double* i(struct b* p) { return p->n; }
void _()
{
    double x;
    [[@spec !id:2]](x);
    [[@spec !id:1]] y;
    [[@spec !id:4]](&y);
    [[@spec !id:5]](i);
}
```

Parametric types instantiation - Examples

- After Stage 3
  - This program is a monomorphized version of the original one, which must be rewritten to C (or whatever internal representation of a compiler).

```
struct a { _Any(_) m; };
[[@spec_struct !id:1 !tag:a !subs:tyof(m)->int]]
[[@spec_struct !id:6 !tag:a !subs:tyof(m)->double*]]
[[@spec_struct !id:7 !tag:a !subs:tyof(m)->int*]]
struct b { double* n; };
_Any(t)* f(_Any(t) p) { struct a _With(m:_Any(t)*)* x; return x->m; }
[[@spec_func !id:2 !name:f !subs:t->double !pends:[[@spec !id:6]]]]
[[@spec_func !id:3 !name:f !subs:t->int !pends:[[@spec !id:7]]]]
int g(_Any(t)* p) { f(p->m); return 1; }
[[@spec_func !id:4 !name:g !subs:t->[[@spec !id:1]] !chains:[[@spec !id:3]]]]
int h(double* (*p) (_Any(t))) { return h(p); }
[[@spec_func !id:5 !name:h !subs:t->struct b* !chains:[[@spec !id:5]]]]
double* i(struct b* p) { return p->n; }
void _()
{
    double x;
    [[@spec !id:2]](x);
    [[@spec !id:1]] y;
    [[@spec !id:4]](&y);
    [[@spec !id:5]](i);
}
```

Compiler pluggability and C compatibility

- Monomorphization enables compiler pluggability through syntax rewriting.
  (Yet, a rewrite to C is just an option, which would allow an implementation to be shared.)
- Compatibility
  - With regard to program translation, can be achieved is the use of parametric types is restricted. (See 3.2 for the details.)
  - Not affected: core typing semantics, typing-unrelated semantics, runtime and interoperability, and most of syntax. Programming style/paradigm, along with a "surface" of typing semantics, aren't affected meaningfully.

Prototype

- For a subset of C
- Available at http://www.genericsinc.info/index.php

Thank you!