Proposal for C2y
WG14 N2698

Enabling Generic Functions and Parametric Types in C

Autor: Leandro T. C. Melo - ltcmelo.com, ltcmelo@gmail.com
Date: 2023, February, 19
Proposal category: New feature
Target audience: Programmers in general and compiler/library programmers

Abstract

I present a design of parametric polymorphism through which I propose extending C with parametric types and generic functions. A prototype of this design (for a subset of C) is available.

1 Introduction

It has always been possible to write a "generic function" in C. The typical approaches by which programmers write a "generic function" in C are the macro approach and the `void*` approach (or a combination of the two); but both of them suffer from weaknesses:

- The macro approach is unanalyzable and fragile. By non-analyzability, I mean that it is impossible to syntactically/semantically analyze a macro, as the preprocessor is agnostic to the phrase structure grammar of C. By fragility, I mean that even a well-analyzed macro expansion is subject to a number of pitfalls, as the expansion of a macro parameter is a plain text copy of its argument.
- The `void*` approach is nonuniform and ineffective. By nonuniformity, I mean that this approach can only be realized with types that are pointers. By ineffectiveness, I mean that an object of type `void*` is not dereferenceable: it must be converted "back" so that it can be dereferenced; yet, an incorrect conversion–in spite of being well-typed–will introduce undefined behaviour to the program.

Since C11, it has also been possible to write a "generic function" by the generic selection approach, i.e. with the `_Generic` keyword. Normally, a generic selection is the expansion of a function-like macro (such as a type-generic macro from the `<tgmath.h>` header of C99) and indeed appears as a generic function at the call site. However, programmers must still designate an actual function for each type in the associations of a generic selection; and hence a generic selection is in reality a (non-generic) type selection dispatcher.

Now, I propose extending C with parametric types and generic functions with a design of parametric polymorphism. Parametric types and generic functions, when compared to the macro approach, improve on analyzability and eliminate fragility; when compared to the `void*` approach, bring uniformity and effectiveness.

2 A Design of Parametric Polymorphism

In this section, I present a design of parametric polymorphism in which parametric polymorphism is accomplished via monomorphization. This design does not attempt to be the most expressive design out there, but to achieve maximum expressiveness in a pluggable and compatible way. I present, in Section 2.1, the syntax of parametric types; in Section 2.2, the syntax to request their instantiation; in Section 2.3, monomorphization; and in Section 3, my criterion of pluggability and compatibility.

Notation
- A term, when introduced, is written with italicized font: e.g. *parametric type*.
- A term that corresponds to a new grammar nonterminal is written with dark grey slanted font: e.g. *structure type instantiation specifier*; the nonterminal corresponding to this term is written hyphenated (possibility with abbreviations) with green italicized font: *struct-type-instantiation-specifier*.

2.1 Specifying Parametric Types and Declaring Generic Functions

A type that is specified with a *parametric type specifier* is a *parametric type*. This specifier consists of the new keyword **_Any** followed by a parenthesized identifier. But a parametric type may only be specified in certain contexts; one such context is when the parametric type is the type of a function parameter. A function declared with a parameter of parametric type is a *generic function*.

Example 1: The figure below shows three generic functions, each one of them with a parametrically typed parameter.

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

A parametric type may also be specified in the construction of an array, pointer, or function type of a function parameter; these types, when constructed from a parametric type, become parametric types too. More precisely, an array type constructed from a parametric type is a *parametric array type*; a pointer type constructed from a parametric type is a *parametric pointer type*; and a function type constructed from a parametric type is a *parametric function type*.[1] A parametric type that is not a parametric array, pointer, or function type is a *plain parametric type*. The parametric function type that is specified by the declarator portion of a generic function is a *bound parametric function type*.

Example 2: The figure below shows four generic functions: the parameter of `f` is of plain parametric type; that of `g` is of parametric array type; that of `h` is of parametric pointer type; and that of `i` is of parametric function type. The type of each one of these functions is a bound parametric function type.

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

In the parameter list of a generic function, parametric types with equal identifiers are the same type.

Example 3: The figure below shows two generic functions: in `f`, the parametric type of parameter `p`, the referenced parametric type of the type of parameter `q`, and the parametric type of the parameter in the function type of parameter `r` all are the same type; in `g`, all parametric types are different types.

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

A parametric type may only be specified as the return type–or in the construction of the return type–of a generic function when this parametric type also is specified as the type–or in the construction of the type–of a parameter of this function. (Observe that the notion of a derived type in C encloses parametric array and pointer types, i.e. a parametric type can, as an element type, derive a parametric array type; and a parametric type can, as a referenced type, derive a pointer type. However, due to the aforementioned stipulation imposed on the return type of a parametric function type, the notion of a derived type C does not enclose parametric function types, i.e. a parametric type cannot, as a return type alone, derive a parametric function type.)

Example 4: The figure below shows three generic functions with a parametrically typed return type.

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

A parametric type may only be specified as the type–or in the construction of the type–of an object declared in the body of a generic function when this parametric type also is specified as the type–or in the construction of the type–of a parameter of this function.

[1] There is a stipulation imposed on the return type of a parametric function type; I explain it later, still in this section.

EXAMPLE 5: The figure below shows a generic function with a parametrically typed object in its body.

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

A parametric type that is the type of a function parameter, a return type, or the type of a local object may be qualified.

EXAMPLE 6: The figure below shows four generic functions with **const**-qualified parametric types.

```
void f(const _Any(t) p) {}
const _Any(t) g(const _Any(t) p) { return p; }
void h(const _Any(t)* p) {}
void i(_Any(t)* const p) { const _Any(t) x; }
```

A parametric type may also be specified, with the underscore as its identifier, as the type of a structure member; such a type is a *parametric member type*. A parametric member type may not be specified in the construction of an array, pointer, or function type, and may not be qualified.[2] A structure type constructed from one or more parametric member types is a *parametric structure type*.

EXAMPLE 7: The figure below shows four parametric structure types with their parametric member types.

```
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; };
```

A parametric structure type may not be specified, by a structure specifier that declares a new type, as the type–or in the construction of the type–of a function parameter, a return type, or the type of a local object; but a parametric structure type may be specified as such types by a *structure type instantiation specifier*, which I present in Section 2.2.1.

2.2 Requesting the Instantiation of Parametric Types

A parametric type, like the **void** type, is an incomplete type that cannot be completed. But the compiler can *instantiate* a parametric type upon an *instantiation request* for it. Instantiating a parametric type consists in *substituting* it for a type that is not parametric; such a type is a *ground instantiation* of this parametric type and can be a complete type. I present, in Section 2.2.1, the syntax for requesting the instantiation of parametric structure types; and, in Section 2.2.2, that for requesting the instantiation of bound parametric function types.

2.2.1 Requesting the Instantiation of Parametric Structure Types

The programmer requests the instantiation of a parametric structure type explicitly with a *structure type instantiation specifier*, which consists of a type specifier followed by an *with selection*; which consists of the new keyword **_With** followed by a parenthesized *member substitution list*; which is a comma-separated list where each element is a *member substitution*; which is a colon-separated pair of an identifier and a type name. The type specifier in a *structure type instantiation specifier* must denote, by either a typedef name or a structure specifier, a parametric structure type. The identifier in a *member substitution* must denote a parametrically typed member of the parametric structure type denoted by its enclosing *structure type instantiation specifier*.

A parametric structure type may be specified, by a *structure type instantiation specifier*, as the type–or in the construction of the type–of a function parameter, a return type, or the type of a local object.

EXAMPLE 8: The figure below shows a parametric structure type as the type of parameters of generic functions.

```
struct a { _Any(_) m; };
void f(struct a _With(m:double)* p) {}
void g(struct a _With(m:_Any(t)) p) {}
```

[2] I justify these stipulations in Section 2.3.4.

A request for the instantiation of a parametric structure type is a *structure type instantiation request*; the compiler always fulfills it. Yet, when this request occurs outside of a generic function, it is an *immediate structure type instantiation request*, that is fulfilled in stage 1 of monomorphization, which I present in Section 2.3.1; whereas when this request occurs inside a generic function, it is a *pending structure type instantiation request* that is pending on the instantiation of the parametric function type that is bound to this function–as I explain in Section 2.3.2, the compiler does not always fulfill a request for the instantiation of a bound parametric function type, but, when it does, any structure type instantiation request that was pending on it is indeed requested and fulfilled in stage 3 of monomorphization, which I present in Section 2.3.3.

Example 9: The figure below shows a program with parametric structure types, a generic function, and a non-generic function. The structure type instantiation requests inside `_` are immediate ones; those inside `f` are pending ones.

```
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;
}
```

2.2.2 Requesting the Instantiation of Bound Parametric Function Types

The programmer requests the instantiation of a bound parametric function type implicitly at a *principal instantiation expression*, which is an *instantiation expression* that occurs outside of a generic function; whereas an instantiation expression that occurs inside a generic function is a *subordinate instantiation expression*. An instantiation expression is an expression that consists of an identifier that denotes a generic function as either the called function within a function call or the operand of a cast. A subordinate instantiation expression is *chained* to a principal instantiation expression when the identifier of the function in which it occurs is denoted:

- by said principal instantiation expression; or
- by a subordinate instantiation expression that is chained to said principal instantiation expression.

A request for the instantiation of a bound parametric function type is a *function type instantiation request*; the compiler can fulfill it, but it is not obliged to do so, as I explain in Section 2.3.2. When the instantiation expression of a function type instantiation request is a function call, a function type instantiation request is a *function type instantiation request by function call*; whereas when this expression is a cast, a function type instantiation request is a *function type instantiation request by cast*.

A function type instantiation request, by function call or by cast, is transitive: it encompasses the parametric function type that is bound to a principal instantiation expression and all parametric function types that are bound to subordinate instantiation expressions that are chained to this principal instantiation expression. Therefore, a single function type instantiation request can instantiate multiple bound parametric function types.

Example 10: Figures 1 and 2 show programs with several generic functions and a non-generic function. In both programs, the identifiers of all function calls and casts inside `_` are principal instantiation expressions. But only in the program of Figure 1 the function type instantiation requests at these expressions are fulfilled; in the program of Figure 2, none of them are. I explain the reasoning behind the fulfillment and non-fulfillment of these requests in Section 2.3.2.

```c
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);
}
```

Figure 1: A program where all of the function type instantiation requests are fulfilled.

```c
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);
}
```

Figure 2: A program where none of the function type instantiation requests are fulfilled.

2.3 Monomorphization

A parametric type can have multiple ground instantiations; each of them reveals a form of this parametric type. The compiler accomplishes parametric polymorphism via *monomorphization*, a program transformation that is composed of the following stages:

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

I explain these stages in the upcoming sections. At the end of each one of them, the program in Figure 3 is shown, in a program representation that is implementation-defined (and internal to a compiler), as transformed by the stage being explained.

```
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);
}
```

Figure 3: An illustrative program for the explanation of the stages of monomorphization.

2.3.1 Stage 1: Structure Types Instantiation - 1^{st} Iteration

In this stage, the compiler fulfills immediate structure type instantiation requests. The substitutions with which the compiler instantiates a parametric structure type are stated by the *member substitution lists* of the *with selection* enclosed by the *structure type instantiation specifier* that denotes this type. Given these substitutions, for each *structure type instantiation specifier*, the compiler:

(a) synthesizes a *structure specialization* of the denoted parametric structure type; and
(b) patches said specifier with a reference to the synthesized structure specialization.

Example 11: The figure below shows the program from Figure 3 transformed by stage Structure Types Instantiation - 1^{st} Iteration. `[[@spec_struct !id:1 !tag:a !subs:tyof(m)->int]]` represents a structure specialization and `[[@spec !id:1]]` represents a reference to this specialization.

```
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);
}
```

2.3.2 Stage 2: Function Types Instantiation

In this stage, the compiler attempts to fulfill function type instantiation requests. The substitutions with which the compiler instantiates a parametric function type are computed by type inference in an implementation-defined manner–in Section 4.2, I recommend an algorithm for this computation. If and only if there exist substitutions such that the semantics of the principal instantiation expression to which a parametric function type is bound is not violated, is a function type instantiation request fulfilled. Given these substitutions, for each principal instantiation expression, the compiler:

(a) synthesizes a *function specialization* of the denoted function;
(b) patches said expression with a reference to the synthesized function specialization; and
(c) repeats (a) and (b) for each subordinate instantiation expression that is chained to said expression.

Below, I explain the reasoning for the fulfillment of the function type instantiation requests of Figure 1.

- The type of `f` is `void(*)(_Any(t))`. In line 14 at `f(x)`, `f` is instantiated as `void(*)(int)`: parameter `p` constrains `_Any(t)` as an arbitrary parametric type; argument `x` constrains `_Any(t)` as an integer type that is compatible with `int`. In line 15 at `f(&x)`, `f` is instantiated as `void(*)(int*)`: parameter `p` constrains `_Any(t)` as it does in `f(x)`, but argument `&x` constrains `_Any(t)` as `int*`. In line 16 at `(void(*)(double))f`, `f` is instantiated as `void (*)(double)`. (Observe that `f(*p)` in line 4 occurs inside a generic function and is subordinate instantiation expression.)
- The type of `g` is `void(*)(_Any(t)*)`. In lines 17 and 19 at `g(&x)` and `g(y)`, `g` is instantiated twice as `void(*)(int*)`: parameter `p` constrains `_Any(t)*` as a parametric pointer type; arguments `&x` and `y` constrain `_Any(t)*` as `int*`; `*p` reinforces this constraint and further constrains `_Any(t)` as a type that can represent lvalues (i.e. a type other than `void`). Due to the transitivity of an instantiation request, in lines 17 and 19 at `g(&x)` and `g(y)`, `f` also is instantiated twice as `void (*)(int)`.
- The type of `h` is `void(*)(_Any(t)*)`, the same as that of `g`. In line 20 at `h(y)`, `h` is instantiated as `void(*)(int const*)` rather than as `void(*)(int*)`, which is how `g` is instantiated in line 18 at `g(y)`: in both `h` and `g`, parameter `p`, together with argument `y`, constrain `_Any(t)*` as a pointer to an integer type that is compatible with `int`; but in `h`, `p = &x` strengthens the constraint over `_Any(t)` to a pointer to `const`-qualified `int`.
- The type of `i` is `_Any(t)(*)(_Any(t))`. In line 21 at `i(1)`, `i` is instantiated as `double(*)(double)`: parameter `p`, together with argument `1`, constrain `_Any(t)` as `int`; the return type of `i` also is `_Any(t)` and hence `p + 1.0` constrains `_Any(t)`, but as `double`; since the `int` constraint over `_Any(t)` is not strict, the `double` constraint over it is allowed to subsume the `int` one–the usual arithmetic conversions are performed on `1`.
- The type of `j` is `void(*)(_Any(t)*, _Any(u))`. In line 22 at `j(y, 1)`, `j` is instantiated as `void(*)(int*, int)`: parameter `p`, together with argument `y`, constrain `_Any(t)*` as `int*`; parameter `q`, together with argument `1`, constrain `_Any(u)` as `int`; `*p = x` constrains `_Any(t)` and `_Any(u)` as compatible types and `_Any(t)` as a type that can represent modifiable lvalues. In line 23 at `j(0, 1.0)`, `j` is instantiated as `void (*)(double*, double)`: although `0` is of type `int`, it also is the null pointer constant, which weakens the constraint over `p`; still, argument `1.0` and `*p = x` are sufficient to constrain `_Any(t)*` as `double*` and `_Any(u)` as `double`.
- The type of `k` is `int(*)(_Any(t))`. In line 25 at `k(z)`, `k` is instantiated as `int(*)(struct a*)`: parameter `p`, together with argument `z`, constrain `_Any(t)` as `struct a*`; `p->m` constrains `struct a` to contain a member named `m`; the return type of `k` constrains the type of this member as `int`.
- The type of `l` is `double(*)(struct b _With(m:_Any(t))*)`. In line 27 at `l(w)`, it is instantiated as `double(*)(struct b _With(m:double)*)`: parameter `p`, together with argument `w`, constraint `_Any(_)` of member `m` of `struct b` as `double`.
- The type of `identity` is `_Any(t)(*)(_Any(t))`. In line 28 at `identity(identity)(1)`, it is instantiated twice, once as `int(*)(int)` and once in delay as `int(*(*)(int (*)(int)))(int)`.

Below, I explain the reasoning for the non-fulfillment of the function type instantiation requests of Figure 2.

- The type of `f` is `void(*)(_Any(t)*)`. In line 14 at `f(*x)`, `f` is not instantiated: parameter `p` constrains `_Any(t)*` as a parametric pointer type; however, argument `*x` constrains `_Any(t)*` as an integer type, which is not compatible to a pointer type. In line 15 at `(void (*)(double))f`, `f` is not instantiated: `double` constrains `_Any(t)*` as a floating type, which is not compatible to a pointer type either.
- The type of `g` is `void(*)(_Any(t))`. In line 16 at `g(x)`, `g` is not instantiated: parameter `p`, together with argument `x`, constrain `_Any(t)` as `int*`; however, `p + 1.0` constrains `_Any(t)` as an arithmetic type–given that one operand of the addition is a value of floating type–, whereas `int*` is a pointer type.

- The type of `h` is `void(*)(_Any(t)*)`. In line 18 at `h(y)`, `h` is not instantiated: parameter `p`, together with argument `y`, constrain `_Any(t)*` as `const int*`; however, `*p = x` constrains `_Any(t)` as a type that can represent modifiable lvalues, whereas `const int` is not such a type.
- The type of `i` is `_Any(t)(*)(_Any(t))`. In line 19 at `i(*x)`, `i` is not instantiated: parameter `p`, together with argument `*x`, constrain `_Any(t)` as `int`; `i = p` reinforces this constraint as strict; the return type of `i` also is `_Any(t)` and hence `p + 1.0` constrains `_Any(t)`, but as `double`; however, since the `int` constraint of `_Any(t)` is strict, the `double` constraint over it is disallowed to subsume the `int` one–otherwise, a value of floating type could be truncated toward zero on `i = p`.
- The type of `j` is `void(*)(_Any(t)*, _Any(u)*)`. In line 21 at `j(z, x)`, `j` is not instantiated: parameter `p`, together with argument `z`, constrain `_Any(t)*` as `struct a*`; parameter `q`, together with argument `x`, constrain `_Any(u)*` as `int*`; however, `p = q` constrains `_Any(t)*` and `_Any(u)*` as compatible types, whereas `struct a*` and `int*` are not compatible. (Observe that it would be possible to instantiate `j` as `void(*)(void*, void*)` in line 21 at `j(z, x)`, but, from a principle standpoint, that would defeat the intent of making `j` a generic function.)
- The type of `k` is `int(*)(_Any(t))`. In line 22 at `k(z)`, `k` is not instantiated: `p->n` constrains `_Any(t)` as a structure type that contains a member named `n`; the return type of `k` constrains this member as `int`; however, argument `z` constrains `_Any(t)` as `struct a`, which does not contain such a member. In line 24 at `k(w)`, `k` is not instantiated: `p->n` and the return type of `k` constrain `_Any(t)` as they do in `k(z)`; however, argument `w` constrains `_Any(t)` as `struct b`, whose member named `n` is of type `int*`.
- The type of `l` is `int(*)(struct c _With(m:_Any(t))*)`. In line 26 at `l(v)`, `l` is not instantiated: `p->n` constrains `_Any(t)` as a structure type that contains a member named `m`; parameter `p` constrains this structure type as `struct b _With(m:_Any(t))*`; the return type of `l` constrains `_Any(_)` of member `m` of `struct b` as `int`; however, argument `v` constraints `_Any(_)` of member `m` of `struct b` as `double`.

Example 12: The figure below shows the program from Figure 3 transformed by stages Structure Types Instantiation - 1^{st} Iteration and Function Types Instantiation. `[[@spec_func !id:2 !name:f !subs:t->double]]` (and similar) represents a function specialization, and `[[@spec !id:2]]` (and similar) represents a reference to this specialization.

```
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);
}
```

2.3.3 Stage 3: Structure Types Instantiation - 2^{nd} Iteration

In this stage, the compiler fulfills structure type instantiation requests in the same way that it fulfills type instantiation requests in the stage Type Instantiation and Specialization - 1^{nd} Iteration, but in this 2^{nd} iteration the compiler fulfills pending–instead of immediate–structure type instantiation requests.

Example 13: The figure below shows the program from Figure 3 transformed by stages Structure Types Instantiation - 1^{st} Iteration, Function Types Instantiation, and Structure Types Instantiation - 2^{nd} Iteration. The structure specialization represented by `[[@spec_struct !id:6 !tag:a !subs:tyof(m)->double*]]` was pending on the function specialization represented by `[[@spec_func !id:2 !name:f !subs:t->double]]`; the structure specialization represented by `[[@spec_struct !id:7 !tag:a !subs:tyof(m)->int*]]` was pending on the function specialization represented by `[[@spec_func !id:3 !name:f !subs:t->int]]`.

```
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);
}
```

2.3.4 Parametrically Member Types: Generality, Expressiveness, and Complexity

In this section, I justify why a parametric member type may not be specified in the construction of an array, pointer, or function type, and my not be qualified. I also justify the reason for the underscore as the identifier of parametric member type. My justifications take into account the generality of parametric types, the expressiveness of generic functions, and the complexity of type inference.

Generality of Parametric Types The stipulations above do not interfere with the generality of parametrically member types: despite the brevity of `_Any(_)`, they are as general as any parametric type.

Expressiveness of Generic Functions I believe that the stipulations above do not harm the expressiveness of generic functions; to the contrary: without them, I believe that such expressiveness could be harmed. To justify this belief, let me consider parametric types that are the type of a function parameter, a return type, or the type of a local object. These parametric types, which are not subject to the stipulations in question, can, in my opinion, enhance the expressiveness of a generic function when specified in the construction of an array, pointer, or function type, when qualified, or when correlated through their identifiers. I accredit this enhancement in expressiveness to the fact that function parameters, return types, and local objects occur in the same context, with type specifier close to their use; for instance, the figure below shows two generic functions that are equally general, but I find `g` more expressive than `f`. On the other hand, generic functions and parametric structure types do not occur in the same context, and the specifiers of parametrically member types are distant from their use; on top of that, a *member substitution* is intended as an explicit (and exact) type substitution–it could be unintuitive/surprising to explicitly request a structure type instantiation such as `struct a _With(m:int)` but obtain a specialization of `struct a` where `m` is not of type `int`.

```
int f(_Any(t) p, _Any(u) q) { return q(p); }
int g(_Any(t) p, int (*q) (_Any(t))) { return q(p); }
```

Complexity of Type Inference The stipulations above ensure that the constraint arising for the computation of substitutions is confined to a function; without them, this constraint would stretch over to the specifier of any parametric structure type with a type instantiation request inside a function. Dealing with such a stretched constraint would only be worth it when paired with a gain in generality and/or expressiveness.

3 Pluggability and Compatibility of the Design

My design of parametric polymorphism revolves around pluggability and compatibility. My criterion of pluggability is straightforward; I explain it in Section 3.1. My criterion of compatibility relies on two perspectives: one that is objective–which is about program translation–, and another that is subjective–which is about language look-and-feel–; I explain the former in Section 3.2 and the latter in Section 3.3; in these sections, I refer to an extension of C that is extended with my design of parametric polymorphism (and nothing else) as C^{α}.

3.1 A Criterion of Pluggability

I state the pluggability of parametric polymorphism as the possibility of implementing it through a plugin, without changing the component of a compiler that is responsible for program translation. (Despite this possibility, a compiler may nevertheless opt to implement parametric polymorphism intrinsically by changing such component.) The motivation for a pluggable design of parametric polymorphism is to enable the sharing of a plugin by multiple compilers; the "interface" for such sharing is *syntax rewriting*. Therefore, the program representation that a plugin uses for monomorphization must be rewritable to C syntax as described below.

Rewriting Structure Specializations
The representation of a structure specialization must be rewritten as a new structure specifier whose syntax is almost identical to that of the parametric structure specifier that it specializes, except that the identifier of the new specifier is replaced by a unique identifier that is customized by the compiler and the parametric member types of the parametric structure identifier are substituted, in the new structure specifier, by their ground instantiations. A reference to a structure specialization must be rewritten as a structure specifier whose identifier is the one customized by the compiler for this specialization.

Rewriting Function Specializations
The representation of a function specialization must be rewritten as a new function whose syntax is almost identical to that of the generic function that it specializes, except that the identifier of new function is replaced by a unique identifier that is customized by the compiler, the bound parametric function type of the generic function is substituted, in the new function, by its ground instantiation, and the parametric types in the body of the generic function are substituted, in the new function, by their ground instantiations. A reference to a function specialization must be rewritten as the identifier customized by the compiler for this specialization.

Example 14: The figure below shows the program from Example 13 rewritten to C syntax.

```
struct a__struct_spec_id1__ { int m; };
struct a__struct_spec_id6__ { double* m; };
struct a__struct_spec_id7__ { int* m; };
struct b { double* n; };
double* f__func_spec_id2__(double p)
{
    struct a__struct_spec_id6__* x;
    return x->m;
}
int* f__func_spec_id3__(int p)
{
    struct a__struct_spec_id7__* x;
    return x->m;
}
int g__func_spec_id4__(struct a__struct_spec_id1__* p)
{
    f__func_spec_id3__(p->m);
    return 1;
}
int h__func_spec_id5__(double* (*p) (struct b*)) { return h__func_spec_id5__(p); }
double* i(struct b* p) { return p->n; }
void _()
{
    double x;
    f__func_spec_id2__(x);
    struct a__struct_spec_id1__ y;
    g__func_spec_id4__(&y);
    h__func_spec_id5__(i);
}
```

3.2 An Objective Perspective of Compatibility of Program Translation

I state the compatibility of program translation of C^α with C under a backward and a forward basis: under a backward basis, I state it as the ability of a C^α compiler to correctly translate C source code; under a forward basis, I state it as the ability of a C compiler to correctly translate C^α source code. Backward compatibility is relevant for programmers in general, as it allows them to gradually upgrade their C source code to C^α source code; forward compatibility is mostly relevant for library programmers, as it allows them to write C^α source code that can be included–not only in C^α source code but also–in C source code.

In regards to backward compatibility, C^α is almost fully compatible with C; full compatibility is prevented because of the new C^α keywords **_Any** and **_With**, which can be regular identifiers in C source code. But if these keywords are put aside, then C source code also is C^α source code. Moreover, since these keywords, in C^α source code, follow and are followed by specific syntax–and in fact can be made contextual keywords–a C^α compiler can, with great accuracy, distinguish eventual uses of **_Any** and **_With** as regular identifiers in C source code. Therefore, I consider that, in practice, C^α is fully compatible with C.

In regards to forward compatibility, C^α is restrictedly compatible with C by parametric type erasure. What I suggest with the erasure of a parametric type is that it is preprocessed as **void*** by a preprocessor macro. Of course, an object with type **void*** is restricted in how it is used, e.g. it cannot be dereferenced; and hence C^α source code that aims to be included in C source code must restrict the use of objects of "parametric type". In effect, parametric type erasure falls back to the **void*** approach of Section 1. However, actual parametric types still are advantageous in a scenario of forward compatibility because, in C^α source code, they can emphasize typing violations in a program; I illustrate parametric type erasure and this advantage in the following example.

Example 15: Figure 4 shows how parametric types can be erased with a preprocessor macro; assume that the source code in this figure is that of `lib.h`. Figure 5 shows a client of `lib.h`; when compiled in "forward compatibility mode" by a C compiler, this program will contain a (silent) undefined behaviour; but when compiled by a C^α compiler, a typing error will be diagnosed during translation.

```
#if __STDC_VERSION__ <= 201710L
#define _Any(whatever) void*
#endif

struct a { _Any(_) m; };
int f(_Any(t) p, void (*q) (_Any(t))) { return q(p); }
```

Figure 4: C or C^α source code of a library `lib.h`.

```
#include "lib.h"

void g(void* p) { int* x; x = p; }
void _()
{
  double* y;
  f(y, g);
}
```

Figure 5: C or C^α source code of a client of the `lib.h` library. When compiled by: a C compiler, the program will contain a (silent) undefined behaviour; a C^α compiler, a typing error will be diagnosed during translation.

3.3 A Subjective Perspective of Compatibility of Language Look-and-feel

I do not state what the look-and-feel compatibility of a language is. Instead, I gather the aspects of C that I believe contribute to its look-and-feel and either point out that a given aspect is not at all affected in C^α or that it is not meaningfully affected in C^α. In conclusion, I claim that C^α is look-and-feel compatible with C.

Programming Style/Paradigm Features of languages often are connected with a programming style/-paradigm. For instance, type inheritance and subtype polymorphism often are connected with object-oriented programming, while first-class functions and function composability often are connected with functional programming. I do not perceive these connections with purism, as I believe that a style/paradigm can still be programmed with a language that lacks the features to which it often is connected. That being said, the style/-paradigm to which generic functions and parametric types often are connected is generic programming; and since generic programming already constitutes the style/paradigm of C due to the macro and `void*` approaches of Section 1, I argue that the programming style/paradigm of C is not meaningfully affected in C^{α}.

Runtime and Interoperability Runtime and interoperability of C are not at all affected in C^{α}.

Elementary Semantics Aspects of the semantics of C like the storage duration of objects, linkage and scope of identifiers, evaluation of expressions, visibility of declarations, and execution of statements remain intact in C^{α}. I refer to the semantics concerning these aspects as the elementary semantics of C, which is not at all affected in C^{α}. The typing semantics of C is affected in C^{α} though; I address it in the next paragraph.

Typing Semantics Although parametric polymorphism enriches the "surface" typing semantics of C^{α}, the "core" typing semantics of C remains intact in C^{α}. More precisely, C^{α} does not modify the representation or compatibility of types in C, and does not introduce new built-in types, type constructors, or type relations. Therefore, the core typing semantics of C is not at all affected in C^{α}. Yet, the surface typing semantics of C^{α} is influenced by type inference. In this regard, as an attempt to preserve, in C^{α}, the "look" of C, I appeal to the implicitness of function type instantiation requests; and as an attempt to preserve, in C^{α}, the "feel" of C, I appeal to the notion of quasi types. Together with parametric polymorphism being accomplished via monomorphization, I argue that the overall typing semantics of C is not meaningfully affected in C^{α}. (Observe that a compiler can output monomorphised syntax–just as it outputs preprocessed syntax–as an aid to the programmer to inspect/verify type inference and to assimilate the look-and-feel of C^{α}.)

Syntax The syntax of lexical elements, expressions, and statements of C remain intact in C^{α}; that of declarations is extended, but I argue that not meaningfully affected in C^{α}. C^{α} introduces, as a *type-specifier*:

- a *parametric-type-specifier*, whose production is **_Any (** *identifier* **)** and resembles that of a C *atomic-type-specifier*, **_Atomic (** *type-name* **)**.
- a *struct-type-instantiation-specifier*, which is produced by a typedef name or structure specifier followed by the new *with-selection*, whose production is **_With (** *member-substitution-list* **)** and resembles that of a C generic selection, **_Generic (** *... , generic-assoc-list* **)**; while the production of a *member-substitution*, *identifier* **:** *type-name*, resembles that of a *generic-association*, *type-name* **:** *assignment-expression*.

4 A Prototype of the Design (for a Subset of C)

I have built a prototype of the design that I present and have made it available at `http://www.genericsinc.info`. This prototype does not cover the entire C^{α} language, but a language that I refer to as B^{α}; the analogy between C^{α} and B^{α} is as follows: just as C^{α} extends C with my design of generic functions and parametric types, B^{α} extends a (hypothetical) subset of C that I refer to as B. Although B is just a subset of C, it attempts to be comprehensive about the typing semantics of C. Therefore, I believe that B^{α} is sufficient to demonstrate both the design that I present and its feasibility.

4.1 The B^{α} Language and its Implementation

The lexical grammar of B^{α} borrows from the grammar of C: lexical elements *identifier* and *constant*; keywords **void**, **int**, **double**, **const**, **struct**, **typedef**, and **return**; and punctuators `[`, `]`, `(`, `)`, `{`, `}`, `->`, `&`, `*`, `+`, `,` and `=`. The phrase structure grammar of B^{α} borrows from the grammar of C several nonterminals, but with only a few of their productions and with simplifications in them; to emphasize such variations, I suffix the nonterminals in the phrase structure grammar of B^{α} with -B^{α}. Moreover, B^{α} does not recognize every sentence that belongs to its grammar; while this is true for C as well, I bring this fact to attention because there are sentences that belong to the grammars of both C and B^{α} but which are recognized in C and not in B^{α}; for instance, a coupled "type and object declaration" such as `struct a { int m; } x;` is recognized in C but not in B^{α}: in B^{α}, the declarations would need to be decoupled into `struct a { int m; };` and `struct a x;`. I do not provide a specification for B^{α}; after all, the non-parametric language within B^{α}, B, is a subset of C; and the phrase structure grammar of B^{α}, which lays below, is extracted from that of C. In the sequence, I highlight what I assess to be, in B^{α}, subtle deviations from C that are not perceptible by the grammar of B^{α}.

Expressions

primary-expression-B^{α}*:*
- *identifier*
- *constant*

postfix-expression-B^{α}*:*
- *primary-expression*-B^{α}
- *postfix-expression*-B^{α} `(` *argument-expression-list*-B^{α} `)`
- *postfix-expression*-B^{α} `->` *identifier*

argument-expression-list-B^{α}*:*
- *assignment-expression*-B^{α}
- *argument-expression-list*-B^{α} `,` *assignment-expression*-B^{α}

unary-expression-B^{α}*:*
- *postfix-expression*-B^{α}
- `&` *unary-expression*-B^{α}
- `*` *unary-expression*-B^{α}

additive-expression-B^{α}*:*
- *additive-expression*-B^{α} `+` *unary-expression*-B^{α}

assignment-expression-B^{α}*:*
- *additive-expression*-B^{α}
- *unary-expression*-B^{α} `=` *assignment-expression*-B^{α}

expression-B^{α}*:*
- *assignment-expression*-B^{α}

Declarations

declaration-B^{α}*:*
- *declaration-specifiers*-B^{α} *declarator*-B^{α} `;`

declaration-specifiers-B^{α}*:*
- *storage-class-specifier*-B^{α} *declaration-specifiers*-$B^{\alpha}{}_{\text{opt}}$
- *specifier-qualifier-list*-B^{α} *declaration-specifiers*-$B^{\alpha}{}_{\text{opt}}$

storage-class-specifier-B^{α}
- **`typedef`**

type-specifier-B^{α}*:*
- **`void`**
- **`int`**
- **`double`**
- *parametric-type-specifier*-B^{α}
- *struct-specifier*-B^{α}
- *struct-type-instantiation-specifier*-B^{α}
- *typedef-name*-B^{α}

specifier-qualifier-list-B^{α}*:*
- *type-specifier*-B^{α} *specifier-qualifier-list*-$B^{\alpha}{}_{\text{opt}}$
- *type-qualifier*-B^{α} *specifier-qualifier-list*-$B^{\alpha}{}_{\text{opt}}$

type-qualifier-list-B^{α}*:*
- *type-qualifier*-B^{α}
- *type-qualifier-list* *type-qualifier*-B^{α}

type-qualifier-B^{α}*:*
- **`const`**

struct-specifier-B^{α}*:*
- **`struct`** *identifier* `{` *struct-declaration-list*-B^{α} `}`
- **`struct`** *identifier*

struct-declaration-list-B^{α}*:*
- *struct-declaration*-B^{α}
- *struct-declaration-list*-B^{α} *struct-declaration*-B^{α}

struct-declaration-B^{α}*:*
- *specifier-qualifier-list*-B^{α} *struct-declarator*-B^{α} `;`

struct-declarator-B^{α}*:*
- *declarator*-B^{α}

struct-type-instantiation-specifier-B^{α}*:*
- *struct-specifier*-B^{α} *with-selection*-B^{α}

parametric-type-specifier-B^{α}*:*
- **`_Any`** `(` *identifier* `)`

with-selection-B^{α}*:*
- **`_With`** `(` *member-substitution-list*-B^{α} `)`

member-substitution-list-B^{α}*:*
member-substitution-B^{α}
member-substitution-list , *member-substitution*-B^{α}

member-substitution-B^{α}*:*
identifier : *type-name*-B^{α}

declarator-B^{α}*:*
pointer-$B^{\alpha}{}_{\text{opt}}$ *direct-declarator*-B^{α}

pointer-B^{α}*:*
∗ *type-qualifier-list*-$B^{\alpha}{}_{\text{opt}}$
∗ *type-qualifier-list*-$B^{\alpha}{}_{\text{opt}}$ *pointer*-B^{α}

direct-declarator-B^{α}*:*
identifier
direct-declarator-B^{α} (*parameter-list*-B^{α})

parameter-list-B^{α}*:*
parameter-declaration-B^{α}
parameter-list-B^{α} , *parameter-declaration*-B^{α}

parameter-declaration-B^{α}*:*
specifier-qualifier-list-B^{α} *declarator*-B^{α}
specifier-qualifier-list-B^{α} *abstract-declarator*-$B^{\alpha}{}_{\text{opt}}$

abstract-declarator-B^{α}*:*
pointer-B^{α}
pointer-$B^{\alpha}{}_{\text{opt}}$ *direct-abstract-declarator*-B^{α}

direct-abstract-declarator-B^{α}*:*
(*direct-abstract-declarator*-B^{α})
direct-abstract-declarator-B^{α} (*parameter-list*-B^{α})

type-name-B^{α}*:*
specifier-qualifier-list-B^{α} *abstract-declarator*-$B^{\alpha}{}_{\text{opt}}$

typedef-name-B^{α}*:*
identifier-B^{α}

type-and-type-synonym-declaration-list-B^{α}*:*
declaration-B^{α}
type-and-type-synonym-declaration-list-B^{α} *declaration*-B^{α}

Statements

statement-B^{α}*:*
expression-statement-B^{α}
return-statement-B^{α}

compound-statement-B^{α}*:*
{ *block-item-list*-B^{α} }

block-item-list-B^{α}*:*
block-item-B^{α}
block-item-list-B^{α} *block-item*-B^{α}

block-item-B^{α}*:*
declaration-B^{α}
statement-B^{α}

expression-statement-B^{α}*:*
expresssion-B^{α} ;

return-statement-B^{α}*:*
return *expression*-B^{α} ;

External definitions

base-translation-unit-B^{α}*:*
type-and-type-synonym-declaration-list-B^{α}
function-definition-list-B^{α}

function-definition-list-B^{α}*:*
function-definition-B^{α}
function-definition-list-B^{α} *function-definition*-B^{α}

function-definition-B^{α}*:*
specifier-qualifier-list-B^{α} *declarator*-B^{α} *compound-statement*-B^{α}

As I mention in the previous paragraph, there are, in B^{α}, subtle deviations from C that are not perceptible by the grammar above. Let me highlight them. In B^{α}:

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

Figure 6: The program from Figure 1 adjusted to B^{α}.

- a *type-and-type-synonym-declaration-list*-B^{α} only recognizes a *declaration*-B^{α} of a type or of a type synonym; e.g. both the declarations `struct a { int m; };` and `typedef struct a a_t;` are recognized.
- a *declaration*-B^{α} is not recognized when it is coupled to a *struct-specifier*-B^{α} with a *struct-declaration-list*-B^{α}, i.e. a type declaration; e.g. neither of the declarations `typedef struct a { int m; } a_t;` or `struct a { int m; } x;` are recognized (to be recognized, the type declaration must be decoupled).
- a *direct-declarator*-B^{α} `(` *parameter-list*-B^{α} `)` is only recognized in a *function-definition*-B^{α}, i.e. function prototypes are not recognized; e.g. the declaration `int f(int p);` is not recognized.
- a `void` *type-specifier* is not recognized in the *specifier-qualifier-list*-B^{α} of a *function-definition*-B^{α}, i.e. functions must have a "valued return"; e.g. the function `void f(int p){}` is not recognized.
- a *compound-statement*-B^{α} is only recognized when its last *statement*-B^{α} is a *return-statement*-B^{α} (this item is complementary to the previous one); e.g. the function `int f(int p){}` is not recognized.

These deviations are not inherent "limitations" of B^{α} but rather incidental to the brevity and conciseness of its implementation. Even so, all examples in this proposal are recognized in B^{α} as long as: array types are adjusted to pointer types, casts are adjusted to function calls, and `void` return type of functions are adjusted to be non-`void`; for instance, Figure 6 shows the program from Figure 1 with these adjustments.

4.2 An Algorithm for Type Inference

There is a type inference algorithm that can be used to compute–when existing–the substitutions that instantiate a parametric function type. This algorithm, which is implemented for my prototype of B^{α}, builds on the techniques presented in the works *Inference of Static Semantics for Incomplete C Programs* [2] and *Type Inference for C: Applications to the Static analysis of Incomplete Programs* [3]; I brief about it below.

Constraint-based Type Inference via Unification In constraint-based type inference via unification, type inference is reduced to a constraint problem for which a solution is to be discovered via unification. A solution that fulfills a function type instantiation request are substitutions that unify *hinted types*, *target types*, and *implied quasi types* that arise as a constraint per pair of instantiation expression and generic function. A hinted type is the type of an argument of the function call in a function type instantiation request by function call; and it is a type from the type name of the cast in a function type instantiation request by cast. A target type is the type of a parametrically typed parameter that corresponds to an argument whose type is a hinted type. An implied quasi type is a *quasi type* that, in the body of a generic function, is the type of an expression that (directly or indirectly) involves either a parametrically typed parameter of this function or a parametrically

typed object that is local to this function. A quasi type is a parametric type that, despite its parametricity, can be classified into a type group like that of integer, arithmetic, scalar, or pointer types. If and only if, for each pair of instantiation expression and generic function in a function type instantiation request, the constraint arising from hinted, target, and implied quasi types is satisfiable, can the compiler fulfill this request; and types are instantiated in delay, i.e., a bound parametric function type only is instantiated once every type that is parametrically typed by this bound parametric function type is instantiated.

5 Related Proposals

- N2632 Type inference for variable definitions and function returns
- N2638 Improve type generic programming
- N2853 The void-which-binds: type-safe parametric polymorphism
- N2953 Type inference for object definitions

6 Normative Changes

The suggested wording is based on ISO-Standard [1].

5.1.1.1 Program structure

At §1.

A C program need not all be translated at the same time... After preprocessing, a preprocessing translation unit is called a ~~translation unit~~ *base translation unit*. A base translation unit undergoes *monomorphization*, a syntactic transformation that eliminates polymorphic syntax. After monomorphization, a base translation unit is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries...

5.1.1.2 Translation phases

At §1, phase 7.

White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens, which comprise a base translation unit, are monomorphized,[3] syntactically and semantically analyzed and translated as a translation unit.

6.2.5 Types

Between §19 and §20.

There is a *parametric type*, designated as **_Any (** *identifier* **)**; it is an incomplete type that cannot be completed, but that can be *instantiated* as a type that is complete (see 6.11).

At §20, item *union type*.

A *union type* describes an overlapping nonempty set of member objects... A parametric type may not derive a union type.

At §20, item *function type*.

A *function type* describes a function with specified return type... A parametric type may only derive a function type when at least one parameter of such a function type is of parametric type or of a type derived from a parametric type.

6.7.2 Type specifiers

At §1.

Syntax

type-specifier:

void
char
short

[3] When a base translation unit does not contain polymorphic syntax, monomorphization may be skipped.

int
long
float
double
signed
unsigned
_Bool
_Complex
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name
parametric-type-specifier
struct-type-instantiation-specifier

6.7.2.5 Parametric type specifier

Syntax

1 *parametric-type-specifier:*
_Any (*identifier*)

Constraints

Semantics

6.7.11 Structure type instantiation specifier

Syntax

1 *struct-type-instantiation-specifier:*
struct-or-union-specifier with-selection
typedef-name with-selection
with-selection:
_With (*member-substitution-list*)
member-substitution-list:
member-substitution
member-substitution-list **,** *member-substitution*
member-substitution:
identifier **:** *type-name*

Constraints

Semantics

6.11 Monomorphization

References

[1] ISO-Standard. ISO/IEC 9899:2018 - The C programming language, 2018.

[2] L. T. C. Melo, R. G. Ribeiro, M. R. de Araujo, and F. M. Q. Pereira. Inference of static semantics for incomplete c programs. *Proc. ACM Program. Lang.*, 2(POPL):29:1–29:28, Dec. 2017. ISSN 2475-1421. doi: 10.1145/3158117. URL `http://doi.acm.org/10.1145/3158117`.

[3] L. T. C. Melo, R. G. Ribeiro, B. C. F. Guimarães, and F. M. Q. a. Pereira. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Trans. Program. Lang. Syst.*, 42(3), Nov. 2020. ISSN 0164-0925. doi: 10.1145/3421472. URL `https://doi.org/10.1145/3421472`.