Source Language Reference

6th October 2023

Contents

1 Type System [types]

1.1 Variants [types.variants]

1.1.1 Definition [types.variants.def]

Source does not have a dedicated union or variant type. Rather, struct types may contain variant clauses introduced by the `variant` keyword. The following example shows a struct type with two variant clauses:

```
struct foo {
    i64 a;
    variant bar { i64 c; i64 d; }
    variant baz { i32 d; }
};
```

The semantics of this type is that it contains a field a of type `i64`, and either

- a field `c` of type `i64` and a field `d` of type `i64`, or
- a field `d` of type `i32`.

TODO: Variant groups (e.g. ast node vs ast type)

1.1.2 Variant Storage [types.variants.storage]

Each variant clause is itself a structure type and may contain any number of fields or variant clauses. All variant clauses of a structure type are stored in overlapping memory (called the *variant storage*); that is, no matter which variant an instance of such a type stores, the memory occupied by the instance is always the same.

The variant storage behaves as though it were a `u8[N]`, where `N` is the size of the largest variant clause plus the size of the variant index if applicable (see next section), and is aligned to the largest alignment amongst all variant clauses (or to that of the variant index if it is placed inside the storage and if its alignment is greater than that of any of the variant clauses).

The variant storage of a type is always placed last after any non-variant fields of the type, irrespective of where the variant clauses are declared in the type definition.

1.1.3 Variant Index [types.variants.index]

To determine which variant an instance of a variant type stores, a variant index is used. The variant index is a non-negative integer that has a different value for each variant clause of the type. The actual value of the index for any given variant clause[1] is implementation-defined and can be accessed using the `::index` metaproperty of the corresponding variant clause (e.g. `foo::bar::index` is the index of the `bar` variant clause of the `foo` type).

The type of the variant index shall be the smallest unsigned integer type that can represent the number of variant clauses of the type, minus one. For example, if a type has 256 variant clauses, the variant index shall be of type `u8`. If a type has 257 variant clauses, the variant index shall be of type `u16`, etc. If there is no such type, the program is ill-formed.

The variant index is a compiler-generated field that shall normally be placed at the beginning of the variant storage as though each variant clause had the index as its first field. However, if the containing structure type contains a contiguous sequence of padding bytes of sufficient size and alignment to hold

[1] Excepting that of a void-variant-clause [types.variants.empty].

the variant index, it is placed in the first such occurrence of bytes instead; this includes padding bytes that may be added after the variant storage at the end of the type.

A user-defined variant index may be declared by declaring a variable of unsigned integer type of sufficient size to hold the number of variant clauses of the type, minus one, and annotating it with the `[[variant\_index]]` attribute. In that case, no variant index is generated by the compiler and the user-defined variant index is used instead. If more than one field of the type is annotated with the `[[variant\_index]]` attribute, the program is ill-formed.

1.1.4 Variant Count [types.variants.count]

The number of variant clauses of a type can be accessed using the `::variant_count` metaproperty of the type (e.g. `foo::variant_count` is the number of variant clauses of the `foo` type). It is an error to attempt to access the value of this metaproperty on a type that is not a structure or variant type.

1.1.5 Initialisation and Empty variant [types.variants.empty]

The variant stored in a default-initialised instance of a variant structure type is implementation-defined. Note that by default, a variant cannot be 'empty'; that is, it must always store a value of one of its variant clauses. As a special case, an explicit void-variant-clause can be created by adding a `variant void` 'field' to the structure. If such a field is present, it will be the default variant stored in a default-initialised instance of the type, and its variant index is always 0. For example:

```
struct maybe_int {
    variant int { i64 value; }
    variant void;
}
```

The semantics of this type is that it contains either a field `value` of type `i64`, or nothing at all. A default-initialised instance of this type will always store the void variant.

Multiple instances of a void-variant-clause in the same structure are ill-formed. Note that a variant clause with no members (e.g. `variant empty {}`) is *not* a void-variant-clause.

1.1.6 Accessing the Variant Type and Storage [types.variants.variant]

The type of all variant clauses can be accessed using the by a metadata expression using the `variant` keyword. For example, `foo::variant` is conceptually equivalent to:

```
struct foo::variant {
    variant bar { i64 c; i64 d; }
    variant baz { i32 d; }
}
```

The variant storage can be accessed by means of a member access expression whose member is the `variant` keyword. For example, the following code declares a variable storage of type `foo::variant` and assigns it the value of the variant storage of `v`:

```
var storage = v.variant;
```

1.1.7 Operator `is` [types.variants.is]

The `is` operator can be used to check whether a value contains a particular variant. For example, the following code checks whether a value `v` of type `foo` contains a variant of type `bar`:

```
if v is bar {
    /// ...
}
```

Note that the expression `expr is bar` is ill-formed if *expr* is not an expression of structure type that contains a variant clause of type `bar`.

1.1.8 Operator `as` [types.variants.as]

The `as` operator can be used to extract a particular variant from a value. For example, the following code creates a variable `bar_value` that holds the variant value of the bar a variable `v` of type `foo`. That is, the type of `bar_value` is `(i64, i64)`:

```
var bar_value = v as bar;
```

Note that the expression `expr a bar` is ill-formed if *expr* is not an expression of structure type that contains a variant clause of type `bar`.

If `bar` is not the active variant, the value of `bar_value` will be the first `foo.bar` :: `size` bytes of the variant storage of `v`. This means that type punning with variants is possible.

1.2 References [types.ref]

Reference types are denoted by the `&` type qualifier. The value of a reference type is the address of an object; references always point to valid objects and can never be `nil`. Unlike references in other languages, they can, however, be reassigned. For ease of use, references are subject to several implicit conversion, depending on the context in which they are used.

1.2.1 Implicit Dereferencing [types.ref.autoderef]

Rvalues of reference type can be implicitly converted to lvalues of the referenced type; that is, if `T` is a type, then an rvalue `T&` is implicitly convertible to an lvalue of type `T`. This conversion is called *implicit dereferencing* or *autodeferencing*.

1.2.2 `is` and `cond` [types.variants.cond]

The body of an *if-expression* whose condition is an *is-expression* is an *implicit with-expression* bound to the value of the `bar` variant of `foo`:

```
if v is bar then print(c + d);
```

That is, the members of the `bar` variant are in scope. Furthermore, a *cond-expression* can be used in a similar manner for pattern matching:

```
cond v is {
    bar: print(a + c + d);
    baz: print(a + d);
}
```

The body of each *cond-case* (excepting the *else-case*) of such a *cond-expression* is an *implicit with-expression* bound to the value of the corresponding variant. Moreover, the entire *cond-expression* is an *implicit with-expression* bound to the value of `v`; this pulls in all regular fields of `foo` as well. As with all *cond-expressions* using `is`, the `fallthrough` keyword is not allowed.

1.3 Type Equality [types.equality]

Equality of types (denoted with $=$) is an equivalence relation that is defined as follows:

- Every type is equal to itself.
- Two array or vector types are equal if they are of the same kind and their element types are equal and they have the same dimension.
- Two slice, range, or *pointer-like* types are equal if they are of the same kind and their element types are equal.
- Two tuple types τ', τ'' are equal if they have the same number of elements, and for their element types τ'_i, τ''_i, it holds that $\tau'_i = \tau''_i$.
- Two function types ϕ', ϕ'' are equal if
  - they have the same number of parameters, and
  - for their parameter types ϕ'_i, ϕ''_i, it holds that $\phi'_i = \phi''_i$, and
  - their return types are equal, and
  - both are variadic or neither one is.
- Two closure types are equal if their function types are equal.

2 Expressions [expr]

2.1 Value Category [expr.value]

Every expression has an associated *value category* that determines whether it is an *lvalue* or an *rvalue*. The value category is uniquely determined by the expression kind and the value category of its operands, after any implicit conversions, if applicable. Note that an expression's value category is independent of its type.

An *rvalue* is a temporary object; it may, but need not, live in memory, and as such, it is invalid to bind a reference to an rvalue. An *lvalue* is an object that has a memory address, and references can bind to lvalues and lvalues only. Note that not all lvalues can be modified.

Below is an exhaustive list of all expressions in the language and their value categories. If an expression is missing, then this is a defect in the specification and should be reported.

2.1.1 Lvalue Expressions [expr.value.lvalue]

The following expressions are lvalues:

- *procedure-declarations* [expr.proc], whether named or not.
- *variable-declarations* [expr.var].
- *non-empty-block-expressions* [expr.block] that are not empty and whose *last-expression* is an lvalue.
- *name-expressions* [expr.name] that denote a variable or parameter.
- *member-access-expressions* [expr.member] whose left-hand-side operand is an lvalue of structure type.
- *unary-prefix-expressions* with operator $\star$ [expr.unary.deref].
- *binary-expressions* with assignment operators [expr.binary.assign], but *not* operator $\Rightarrow$ [expr.binary.refassign

2.1.2 Rvalue Expressions [expr.value.rvalue]

The following expressions are rvalues:

- *invoke-expression*s, including such as return a value of reference type [expr.invoke].
- *block-expressions* [expr.block] that are empty or whose *last-expression* is an rvalue.
- *name-expresisons* that denote a procedure or type [expr.name].
- *member-access-expressions* [expr.member] whose left-hand-side operand is an rvalue or not of structure type.[2]
- *literal-expression*s, including *string-literal*s [expr.literal].
- *binary-expressions* [expr.binary] with non-assignment operators [expr.binary.assign] or operator ⇒ [expr.binary.refassign].
- *type*s [expr.type].

2.2 Operand Value Categories [expr.value.operands]

Every expression mandates a certain value category and type for each of its operands. Let P^i denote the mandated operands for an expression E, A^i the actual operands; given an abf expression e, let $C(e)$ be the value category of e, and $T(e)$ the type of e. The following algorithm is performed for each A^i:

1. If $T(A^i) \neq T(P^i)$, and there is an implicit conversion of A^i (!) to $T(P^i)$, perform that conversion. If there is no such conversion, the program is ill-formed.
2. If $C(A^i) \neq C(P^i)$, and $C(A^i)$ is an lvalue, perform lvalue-to-rvalue-conversion; otherwise, the program is ill-formed.[3]

Example

```
proc by_ref (i32&) {}
proc by_val (i32) {}

i32 i;
by_val i; /// OK: same type; lvalue-to-rvalue conv. from lvalue i32.
by_val 4; /// OK: const-expr conv. from rvalue int to i32.
by_ref i; /// OK: conv. from lvalue i32 to rvalue i32&.
by_ref 4; /// ERROR: No conv. from rvalue int to i32&.
```

2.2.1 Operand Value Constraints by Expression Kind [expr.value.constraints]

If an expression is not listed below, it either has no operands (e.g. [expr.unreachable]) or no constraints associated with its operands in the general case (e.g. [expr.block]).

- For an *invoke-expression*, operand type constraints are determined via overload resolution [expr.overload]. All operands must be rvalues.
- For a *variable-declaration*, if the declaration has an initialiser, the type of the initialiser must be an rvalue of the same type as the type of the declared variable.

[2] This also means that you cannot e.g. assign to the `.data` or `.size` member of a slice or rebind them.

[3] Note that an rvalue of reference type would have already been converted to an lvalue of the element type in the previous step.

- For a *member-access-expression*, the LHS operand must be of slice, range, or structure type or of a pointer-like type thereto.
- For a *unary-prefix-expression*,
  - with operator `*`, the operand must be of a pointer-like type.
- For a *binary-expression*,
  - with an arithmetic operator, the operands must be rvalues of integral type;
  - with a comparison operator, the operands must be rvalues of type `bool`;
  - with a value-assignment operator (e.g. `=`), the LHS operand must be an lvalue of the same type as the RHS. The RHS must be an rvalue;
  - with the reference-assignment operator `=>`, the LHS must be an lvalue of reference type and the RHS must be an rvalue of the same type. [expr.binary.refassign]
- For a procedure body introduced with `=`, the type of the body must be an rvalue of the same type as the procedure's return type.

2.3 Binary Expressions [expr.binary]

Binary expressions have a LHS and a RHS that are combined with an operator. The semantics of a binary expression depend on the operator and sometimes on the types and value categories of the operands.

2.3.1 Reference Reassignment [expr.binary.refassign]

The reference reassignment operator `=>` is a special binary operator that is used to rebind references. Unlike the value assignment operator `=`, there are no compound variants of this operator. The yield of the expression is an lvalue to the reassigned reference. The LHS of the expression must be convertible to an lvalue of reference type T, and the RHS to T.

The interaction of this expression with value categories and autodereferencing is a bit complicated. To elaborate, we first define the notion of the reference depth of a type. Let $\mathit{Elem}(t)$ be the element type of a reference type t. The reference depth $D(t)$ of a type t is 0 if t is not a reference type, and $1 + D(\mathit{Elem}(t))$ otherwise.

The algorithm below illustrates the exact semantics of this operation. Let l be the LHS and r the RHS. Let $T(e)$ be the type of an expression e. For brevity, we may write $D(e)$ for $D(T(e))$ if e is defined to be an expression.

1. If $D(l)$ is 0, the program is ill-formed.
2. If $D(l)$ is 1, and l is an rvalue, the program is ill-formed.
3. If l is an rvalue, perform reference-to-lvalue conversion on l, yielding a new l.[4]
4. If $D(l) < D(r)$, perform lvalue-to-rvalue conversion on r, yielding a new r; then, perform dereferencing followed by lvalue-to-rvalue conversion $D(r) - D(l)$ many times, yielding a new r. Go to step 6.[5]

[4] The reference depth of l in that case is reduced by one, e.g. rvalue `int&&` becomes lvalue `int&`.
[5] This removes that many levels of indirection from the RHS, yielding an rvalue.

5. Otherwise, if $D(l) > D(r)$, perform lvalue-to-rvalue conversion followed by dereferencing $D(l) - D(r)$ many times on l, yielding a new l.[6]

6. If there is no implicit conversion from $T(r)$ to $T(l)$, the program is ill-formed.

7. Otherwise, perform that conversion, followed by lvalue-to-rvalue conversion, yielding y, and store y in the lvalue l; the result of this operation is the lvalue l. Skip all remaining steps.

2.4 Names [expr.name]

2.4.1 Definitions [expr.name.def]

An *unqualified-name* is an `<identifier>` token that is not part of a qualified name [expr.name.def].

A *qualified-name* is a sequence of one or more unqualified names separated by `.` tokens. Note that identifiers separated by the metaproperty access operator `::` do not constitute qualified names. E.g. in the sequence

```
foo.bar::baz.quux
```

there are two qualified names: `foo.bar` and `baz.quux`. Thus, this is parsed as:

```
(foo.bar)::(baz.quux)
```

Note: `<identifier>` tokens created by the `__id` keyword are never qualified names, even if they contain `.` tokens.

2.5 Name Lookup [expr.name.lookup]

If the name is not a *qualified-name*, unqualified name lookup is performed. Otherwise, qualified name lookup is performed. The algorithm for unqualified name lookup is as follows:

1. Let `n` be the text value of the *unqualified-name*; let `ref` be null. If, during the execution of any of the steps below, the value of `ref` is changed,

   (a) in a context where shadowing is enabled [expr.name.shadow], then, if the current step is a loop ('For each'), run the current iteration of that loop to completion; then, skip to the last step of this algorithm;

   (b) in a context where shadowing is disabled, if `ref` is already non-null, the program is ill-formed.

2. If we are currently analysing an *enum-declaration*, let `d` be that declaration. If `d` contains an enumerator `e` whose name matches `n`, set `ref` to an *enumerator-reference* to `e`. If `e` is not yet complete [expr.enum], the program is ill-formed.

3. If there is an expected type [expr.expected], let `t` be that type.

   (a) If `t` is an enum type containing an enumerator `e` whose name matches `n`, replace the name with an *enumerator-reference* to `e`. Note that it is irrelevant whether `e` is complete at this time or not.

   (b) If `t` is a record type containing a variant clause `v` whose name matches `n`, replace the name with a *type-reference* to `v`.

4. For each scope in the scope tree starting at the scope containing the name, and ending at the global scope:

[6] This removes that many levels of indirection from the RHS, yielding an *lvalue*, unlike in the previous step.

   (a) If the scope contains a declaration `d` whose name matches `n`, replace the name with a reference to `d`.
   (b) If the scope contains an overload set whose name matches `n`, replace the name with a reference to that overload set.
   (c) If a *with-expression* `w` is in scope and complete [expr.with], and `w` contains a member whose name matches `n`, replace the name with a *member-access* from `w` to `n`.

5. If template instantiation is currently taking place [expr.template.inst], for each template `t` on the instantiation stack, if there is a template parameter of `t` that matches `n`, replace the name with a copy of the template argument value bound to the parameter in the instantiation.

6. If `n` is the name of an imported module, replace it with a *module-reference* to that module.

7. If the translation unit is a logical module [module.def] and its name is `n`, replace the name with a *module-reference* to that module.

8. For each open imported module [module.def], check if that module exports a declaration whose name matches `n`. If so, replace the name with a reference to that declaration.

9. Replace the name with `ref`. If `ref` is null, the program is ill-formed.

The algorithm for qualified name lookup is as follows:

1. Let `n` be the text value of the *qualified-name*.

2. If there is an imported module whose name matches `n`, replace the *qualified-name* with a *module-reference* to that module and stop.

3. Otherwise, split `n` at the last occurrence of `.` into `n1` and `n2`. Construct a *member-access* from `n1` to `n2` and perform name lookup on `n1` and `n2`.

4. Perform member lookup [expr.member-access] on the result of the previous step.

2.6 Type Conversion [expr.convert]

During semantic analysis, we often have to convert expressions from one type to another; the algorithm below describes how this conversion is performed. This algorithm, named Convert, both checks if the conversion is possible and replaces the expression to be converted with either a *cast-expression* or the converted value.

In some cases, we only want to check if a conversion is possible, but not actually perform the conversion. This can be accomplished by skipping any parts of steps annotated with square brackets. This variation of the algorithm is called Try-Convert.

The algorithm returns an integer *score* that indicates how good the conversion is, with `0` being optimal, a negative value indicating failure, and a higher positive value indicating a higher cost. This score is used for overload resolution [expr.overload].

- Let f (‘from’) be the type of the expression E to be converted, and t (‘to’) the type to convert to.

- If $f = t$, return 0.

- If t is `void`, return 0.

- If f is `noreturn`, [replace E with a *cast-expression* of E to t and] return 0.

- If t is `type` and E is a type, evaluate E as a constant expression, yielding R; [replace E with R and] return 0.
- If E is an lvalue, and t is a reference type whose element type is f, [replace E with a *cast-expression* of E to t and] return 0.
- If f and t are weak pointer-like types of the same kind, then,
  - if t is a void pointer, [replace E with a *cast-expression* of E to its corresponding void pointer type and] return 1; or
  - if the element type of f is an array type whose element type is equal to the element type of t, [replace E with a *cast-expression* of E to t and] return 1 (i. e. a pointer or reference to an array is implicitly convertible to a pointer or reference to its first element); or
  - otherwise, return -1.
- If f is a reference type and t a pointer type, and the element types of f and t are the same, [replace E with a *cast-expression* of E to t and] return 1.
- If f and t are integer types, then,
  - if f is the integer literal type, and t is `isz`, [set the type of E to `isz` and] return 0; or
  - if E is a constant expression, evaluate E as a constant expression, yielding R; if the value of R is representable by t, [replace E with R and] return 1; otherwise, return -1; or,
  - if E is not a constant expression, and f is narrower than t, then—unless f is signed and t isn't—[replace E with a *cast-expression* of E to t and] return 1; otherwise, return -1; or,
  - otherwise, return -1.
- If either type is a slice with element type e, and the other is a tuple type containing two elements, namely pointer to e and `isz`, [replace E with a *cast-expression* of E to t] return 1.
- If either type is a tuple type and the other a structure type that is not packed and contains no variants, and the tuple contains the same number of elements as the structure has fields, and each element of the tuple is convertible to the corresponding field of the structure, and all fields of the structure have their natural alignment, [replace E with a *cast-expression* of E to t] return 1.
- If f and t are integer ranges, and E is a constant expression, evaluate E as a constant expression, yielding R; if the start and end value of R is representable by t, [replace E with R and] return 1; otherwise, return -1.
- If f is a function type and t a closure type whose procedure type is equal to f, [replace E with a *cast-expression* of E to t] return 1.
- Otherwise, return -1.

2.7 Overload Resolution [expr.overload]

To resolve an overloaded callee of a *call-expression* as well as any arguments of the call that are also overload sets, the following algorithm is used. In the algorithm below, upper indices are associated with candidates of overload sets, and lower indices with parameters and arguments.

Let O be the overload set containing candidates C^i, each with parameters C^i_j; let A_j be the arguments of the call (note: the arguments may be unresolved overloads themselves). Let $\|C^i\|$ be the number of parameters of C^i and $\|A\|$ the number of arguments of the call expression. For each candidate C^i, let $A^i_j := \{A_j : j \leq \|C^i\|\}$ be the *non-variadic* arguments for C^i. Then:

1. Mark all C^i as viable.
2. For each candidate C^i:
   (a) If $\|C^i\| > \|A\|$, i. e. the function takes more parameters than there are arguments, mark C^i as non-viable and continue with the next candidate.
   (b) If $\|C^i\| < \|A\|$ and C^i is not variadic, mark C^i as non-viable and continue with the next candidate.
   (c) For each A^i_j that is not an overload set, let $s^i_j := \textsc{Try-Convert}(A^i_j, C^i_j)$ as defined in [expr.convert]. If $s^i_j = -1$, mark C^i as non-viable and continue with the next candidate.
   (d) For each A^i_j that is an overload set:
      i. Let Γ^k be the candidates of A^i_j.
      ii. For each Γ^k, let $\sigma^k := \textsc{Try-Convert}(\Gamma^k, C^i_j)$.
      iii. If all $\sigma^k = -1$, mark C^i as non-viable and continue with the next candidate.
      iv. Otherwise, let $s^i_j := \min\{\sigma^k : \sigma^k \neq -1\}$, and let γ^i_j be the candidate Γ^k with $\sigma^k = s^i_j$. If there are multiple possible γ^i_j, mark C^i as non-viable and continue with the next candidate.
   (e) Let $s^i := \sum_j s^i_j$.
3. If there is no viable C^i, the program is ill-formed.
4. Let μ be the index such that $s^\mu = \min\{s^i\}$ and C^μ is viable. If there are multiple possible μ, the program is ill-formed.
5. For all A_k with $\|C^\mu\| < k \leq \|A\|$ that are overload sets:
   (a) If A_k contains more than one candidate, the program is ill-formed.
   (b) Otherwise, resolve A_k to its only candidate.
6. Resolve O to C^μ
7. Resolve each A^μ_j that is an overload set to its corresponding γ^μ_j as determined in step 2(d)iv.

2.8 Declarations [expr.decl]

2.8.1 Decl-Types [expr.decl.type]

Not every type can occur in a declaration. Thus, in order to make sure that a type is valid, *decl-type decay* is applied to any type to be used in a declaration. The decay is defined as follows:

- If the type is a function type, the decayed type is the corresponding closure type.
- Otherwise, the decayed type is the type itself.

2.9 Scope [expr.scope]

A new scope is opened by

- the global scope;
- a block-expr that is not a delim-expr;

- a static-if-expr;
- an if-expr, encompassing the condition and its body (but not any elif or else branches);
- an elif-expr, encompassing the condition and its body (but not any other elif or else branches);
- an else-expr;
- a while-expr, encompassing the condition and its body;
- a for-expr, encompassing its initialisers and its body;
- a proc-def-expr, encompassing its parameter declarations and its body;
- a cond-expr, encompassing the standard of comparison and the body of each branch;
- a cond-case, encompassing its pattern and its body;
- a with-expr that has a body;
- a defer-expr.

Note that scopes are mainly relevant for three things:

1. The visibility of declarations. A declaration is visible in the scope it is declared in and all scopes nested within it. Declarations inside of expressions that do not open a scope 'float up' to the nearest enclosing scope.
2. Insertion of destructor calls for local variables and parameters. Destructors are called in the reverse order of construction at the end of the scope they are declared in.
3. Placement of defer-exprs. The rules for the eventual placement of defer-expressions are the same as for destructors. Note that defer-expressions 'float up' as well, but since their type is `void`, they are automatically disallowed inside of most expressions where they would otherwise be difficult to handle, such as assert-exprs, the condition of an if-expr, or on either side of the `and`/`or` operators.