Error and Warning reported by Infer 1.5

INFER 1.5 outputs various types of messages as part of the analysis process. Each message is assigned to one of three categories. The two categories: (1) ERROR, and (2) WARNING. Furthermore, each error is assigned to one of the two categories PROVER or CHECKER. The category of PROVER errors is covered by INFER specifications. When INFER 1.5 produces a set of specifications, it effectively shows the absence of run-time memory errors when the procedure is executed in a way that meets the pre-condition. CHECKER represents a weaker type of error not covered by specifications.

6.1 ERROR

INFER 1.5 reports on the following basic types of errors:

1. ARRAY_OUT_OF_BOUNDS_L1 (CHECKER)
2. DANGLING_POINTER_DEREFERENCE (PROVER)
3. DEALLOCATE_STACK_VARIABLE (PROVER)
4. DEALLOCATE_STATIC_MEMORY (PROVER)
5. DIVIDE_BY_ZERO (CHECKER)
6. MEMORY_LEAK (PROVER)
7. NULL_DEREFERENCE (PROVER)
8. RESOURCE_LEAK (PROVER)
9. USE_AFTER_FREE (PROVER)

6.1.1 ARRAY_OUT_OF_BOUNDS_L1

An ARRAY_OUT_OF_BOUNDS_L1 (Level-1) error occurs when it is determined an array access is found to be outside the array's defined range or bounds.

ARRAY_OUT_OF_BOUNDS_L1 represents one of two complex checkers for array bounds errors. The other is ARRAY_OUT_OF_BOUNDS_L2 (Level-2). L1 is a stronger form of error than L2. It occurs when there is direct evidence an array access is out of bounds. L2 is classified as a warning, and occurs when evidence in only indirect, for example, based on possible outcomes involving index operations and loop constructs.

6.1.2 DANGLING_POINTER_DEREFERENCE

A DANGLING_POINTER_DEREFERENCE error occurs when a pointer is de-referenced which does not point to valid memory, such as a pointer de-referenced before initialization or an invalid pointer.

6.1.3 DEALLOCATE_STACK_VARIABLE

A DEALLOCATE_STACK_VARIABLE error is generated when a program attempts to free memory from the stack (e.g. a local variable) rather than the heap; for example, the function

```
void f() { int x; free( &x ); }
```

6.1.4 DEALLOCATE_STATIC_MEMORY

A DEALLOCATE_STATIC_MEMORY error is generated when a program attempts to free memory allocated statically (e.g. a constant string) rather than dynamically on the heap; for example, the function

```
void f() { char *s = "string"; free( s ); }
```

6.1.5 DIVIDE_BY_ZERO

A DIVIDE_BY_ZERO error occurs when a value is divided by 0, as in `x/y` or `x%y` when the value of `y` is zero.

6.1.6 MEMORY_LEAK

A MEMORY_LEAK error occurs when memory allocated with a program or procedure goes out of scope so that it cannot be returned to heap. An example is

```
void f() { int *x = malloc ( sizeof ( int ) ); }
```

where memory is allocated to `x` but not freed before x goes out of scope.

6.1.7 NULL_DEREFERENCE

A NULL_DEREFERENCE error occurs when a pointer, which might be *null,* is de-referenced. An example is

```
void f() { int *x = NULL; int *y = x; *y = 1; }
```

where `y` is assigned to the location given by `x`, which is NULL. The assignment `*y = 1` de-references `NULL` generating the error.

6.1.8 RESOURCE_LEAK

A RESOURCE_LEAK error occurs when a resource, such as *file handle*, is allocated by a program or procedure but never released back to operating system. An example is

```
void f() { FILE* fp = fopen ( "/etc/printcap", "r" ); }
```

where file handle `fp` is allocated (*opened*) but not released (*closed*) before `fp` goes out of scope.

6.1.9 USE_AFTER_FREE

A USE_AFTER_FREE error occurs when an attempt is made to access memory which has been previously freed; for example,

```
void f( int *i ) { free ( i ); *i = 1; }
```

6.2 WARNING

INFER 1.5 reports on the following basic types of warning:

1. ARRAY_OUT_OF_BOUNDS_L2
2. COMPARING_FLOAT_FOR_EQUALITY
3. CONDITION_IS_ASSIGNMENT
4. NULL_TEST_AFTER_DEREFERENCE
5. PRECONDITION_NOT_MET
6. RETURN_VALUE_IGNORED
7. RETURN_EXPRESSION_REQUIRED
8. RETURN_OF_STACK_VARIABLE_ADDRESS
9. RETURN_STATEMENT_MISSING
10. UNARY_MINUS_APPLIED_TO_UNSIGNED_EXPRESSION
11. UNINITIALIZED_VALUE

6.2.1 ARRAY_OUT_OF_BOUNDS_L2

An ARRAY_OUT_OF_BOUNDS_L2 (Level-2) error occurs when it is determined an array access is found to be outside the array's defined range or bounds.

ARRAY_OUT_OF_BOUNDS_L2 represents one of two complex checkers for array bounds errors. The other is ARRAY_OUT_OF_BOUNDS_L1 (Level-1). L1 is a stronger form of error than L2. It occurs when there is direct evidence an array access is out of bounds. L2 is classified as a warning, and occurs when evidence in only indirect, for example, based on possible outcomes involving index operations and loop constructs.

6.2.2 COMPARING_FLOAT_FOR_EQUALITY

A COMPARING_FLOAT_FOR_EQUALITY warning occurs when two floating types are compared for equality (==) or inequality (!=). An example is

```
int f( float x, float y ) { if ( x==y ){ do_something(); } return 0; }
```

The problem is that different compilers and CPU architectures can store floating point results at different precisions, so floating point comparisons may differ depending on the environment in which they are executed. Instead, floating point values should be compared for equality within tolerances based on the machine precision.

6.2.3 CONDITION_IS_ASSIGNMENT

A CONDITION_IS_ASSIGNMENT warning occurs when a Boolean condition is represented by an assignment . An example is

```
int f() { int x,y; … if ( x=y ){ do_something(); } return 0; }
```

Serious problems can arise if the assignment was intended to be only a comparison. If it was meant to be an assignment it is safer to make explicitly outside the if condition.

6.2.4 NULL_TEST_AFTER_DEREFERENCE

A NULL_TEST_AFTER_DEREFERENCE warning occurs when a pointer is first dereferenced and later tested for NULL. An example is

```
int f(int *x) { *x = 3; if(x) return *x; }
```

where pointer `x` is tested by `if(x)` after it has been dereferenced by `*x = 3`.

6.2.5 PRECONDITION_NOT_MET

A PRECONDITION_NOT_MET warning occurs when a function is called but it is not possible to establish a valid pre-condition for the function. For example, suppose a function `f` has been defined as

```
void f (int b) { int *x = malloc(sizeof(int)); if(b==2) free(x); }
```

Infer computes one **precondition** for this function: calling `f` is 'safe' only when parameter `b` has value 2, otherwise the memory allocated to `x` could be leaked. If the function is called with a value other than 2, for example calling `f(1)`, a PRECONDITION_NOT_MET warning is flagged.

6.2.6 RETURN_VALUE_IGNORED

A RETURN_VALUE_IGNORED warning occurs when a function returning a non-void result is called but the result of the function is ignored by the caller.

6.2.7 RETURN_EXPRESSION_REQUIRED

A RETURN_EXPRESSION_REQUIRED warning occurs when a function has non-void return type but does not contain an expression in the return statement.

6.2.8 RETURN_OF_STACK_VARIABLE_ADDRESS

A RETURN_OF_STACK_VARIABLE_ADDRESS warning occurs when a function returns the address of a local variable.

6.2.9 RETURN_STATEMENT_MISSING

A RETURN_STATEMENT_MISSING warning occurs when a function has non-void return type but contains an exit point without a return statement. For example,

```
int f( int b) { if ( b == 2 ) return b; }
```

6.2.10 UNARY_MINUS_APPLIED_TO_UNSIGNED_EXPRESSION

An UNARY_MINUS_APPLIED_TO_UNSIGNED_EXPRESSION warning occurs when a unary minus is applied to an unsigned type, and the result is then used to initialise an object of type long. For example:

```
void f() { unsigned int x=1;  long l=-j;}
```

Depending on the specific platform, if *sizeof (long) > sizeof (int)* then the value of l equates to (UINT_MAX -1). However, if *sizeof(long) ≥ sizeof(int)* then the value of l is -1. This subtle property can lead to array out of bounds errors as well as infinite loops.

6.2.11 UNINITIALIZED_VALUE

An UNINITIALIZED_VALUE warning occurs when an entity such as a variable, struct field, or other addressable memory object is read or accessed before it is assigned a value. An example is

```
int f() { int *x = malloc(sizeof(int)); if(x) return *x; }
```

where pointer `x` is allocated and then its content returned before it has been given a value.