Type systems are crucial tools in the hands of developers. The compiler’s type checking is the first line of defense against programmer error. Unfortunately, developers often not use type systems to their capacity. This is most prevalent with trivial values expressed by fundamental (e.g., `int`) or library (e.g., `string`) types. Developers often resort to embedding the meaning of a variable’s expressed value in its name [1] and not its type. Perhaps the most widely-known example of such weak interfaces is the Mars Climate Orbiter incident [2]. Due to a design misunderstanding, some program modules used different units of measurement conceptually while only communicating in terms of “numbers”, which resulted in the incident. Compilers can catch such mistakes, but only if the program uses types more elaborate than `double`.

**Type migration** is the process of changing the types of program elements. Conventionally, one would design the new types in advance, specify a mapping from old types to the new ones, and perform the migration. If some operations are left undefined, the code does not compile, and the next round begins. If an irrecoverable error is discovered, the algorithm terminates with the error. For example, a function containing two `return` statements that return values of different fictive types would classify as an irrecoverable error.

### Fictive Types

Our approach uses the fictive types (“\(\overline{F}\)”) annotation technique to embed additional information about program elements into the source code. The highlight of annotations is that compiler and tool vendors are allowed to specify their own set. Tools are encouraged to ignore – maybe with an accompanying warning – annotations they do not understand. The following example shows a local variable whose “real” type is `int` and fictive type temperature. Existing compilers and tools interact with the real type only, and the project can be continually re-released, while tools more aware interact with the fictive type. Fictive types express only the “set membership” relation – “SensorTemp is a temperature”.

\[
\text{\((\text{temperature})\)} \quad \text{int} \quad \text{SensorTemp};
\]

### Iterative refactoring process

The refactoring consists of three steps. The propagations step are executed in a saturating fix-point iteration, where the developer is asked on-demand to provide additional input.

### Taint seeding

Consider the previous example wherein the developer decided some variable is a “temperature”. This variable is passed to a function somewhere, and that function returns triple the value, and this is emitted to some output.

\[
\text{int} \quad \text{threshold}(\text{int}) \quad \text{return} \quad 3 \ast \ast T; \}
\]

\[
\text{int} \quad \text{T2} = \text{threshold}();\quad \text{write}(\text{T2});
\]

Code analysis and transformation is executed on the abstract syntax tree (AST). A simplified AST for the previous example – with the type colouring – is shown below.

![AST diagram](image-url)

#### Propagation

The propagation is executed via a modified compiler built upon the LLVM Compiler Infrastructure’s Clang project. It is a monolithic operation where more program elements receive a taint. In Round 1, the propagation tool discovers that the fictively typed variable is passed to the function parameter – a trivial propagation via assignment. This turns threshold into a function taking “temperature”. As each round is as expansive as possible, the type expression \(\ast \ast \text{temperature}\) is investigated, where a recoverable error is generated for the undefined operation.

**Round 2** associates the result of the \(\ast \) operation with the return value of the function, and the assignment of the function’s result to the local variable results in tainting the local variable. The user at this point could decide that write is a library function which they do not wish to change the type of. This results in an explicit type cast away from the new type. As there are no more production rules to take, the algorithm terminates successfully.

\[
\text{int} \quad \text{threshold}(\text{int}) \quad \text{return} \quad 3 \ast \ast \text{F}; \}
\]

\[
\text{int} \quad \text{T2} = \text{threshold}();\quad \text{write}(\text{T2});
\]

### References

A full-length SANER paper in the proceedings with the same title accompanies this poster.

