Interactive Tooling Support for the Migration to Strong Types

Richárd Szalay
INTERACTIVE TOOLING SUPPORT FOR THE MIGRATION TO STRONG TYPES

RICHÁRD SZALAY

DEPARTMENT OF PROGRAMMING LANGUAGES & COMPILERS,
ELTE – EÖTVÖS LORÁND UNIVERSITY,
BUDAPEST, HUNGARY

2023. 05. 10.

Prepared with the professional support of the Doctoral Student Scholarship Program of the Co-operative Doctoral Program of the Ministry of Innovation and Technology financed from the National Research, Development and Innovation Fund.
1 Introduction
   - Strong typing
   - Type migration – by hand

2 Overview
   - Initial taint
   - Propagation
   - Rewriting the code

3 In detail
   - What is a “fictive type”?
   - Code generation
   - Problems Design decisions

4 Tooling & “Demo”
   - Initial setup
   - Propagation
   - “Old” infrastructure
   - Inlining
   - Iterative driver
Introduction
- Strong typing
  What? & How?
  Negative overhead?
  Some existing libraries
- Type migration – by hand

Overview

In detail

Tooling & “Demo”
Snowmass Village

Population: 2863
Elevation: 8410
Incorporated: 1977
Dogs Registered: 210
Total: 13460
Richárd Szalay. **Weak Interfaces → Weak Defences: The Bane of Implicit Conversions in our Function Calls.** C++Now 2021. May 4, 2021. URL: http://youtube.com/watch?v=-UW4tA5r2QE

* case of error.

```cpp
int Classify::MakeNewTemporaryConfig(ADAPT_TEMPLATES Templates,
CLASS_ID ClassId,
    // 1 the first parameter in this range is 'ClassId'
int FontinfoId,
    // 3 after resolving type aliases, type of parameter 'ClassId' is 'int'
int NumFeatures,
    // 2 the last parameter in this range is 'NumFeatures'
        // 5 after resolving type aliases, type of parameter 'NumFeatures' is 'int'
INT_FEATURE_ARRAY Features,
        // 6 3 adjacent parameters for 'MakeNewTemporaryConfig' of similar type are easily
        // swapped by mistake
FEATURE_SET FloatFeatures) {
```
<table>
<thead>
<tr>
<th>Report hash</th>
<th>File</th>
<th>Message</th>
<th>Checker name</th>
<th>Analyzer</th>
<th>Severity</th>
<th>Bug path length</th>
<th>Latest review status</th>
</tr>
</thead>
<tbody>
<tr>
<td>78605d58fc...</td>
<td>.cc</td>
<td>2 adjacent parameters for <code>validateConstraintModifier</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>177deed12cf...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>makeUnconditional</code> of similar type (`llvm::BasicBlock *) are easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6b86ed127d...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>lostFracThroughTruncation</code> of similar type (<code>unsigned int</code>) are easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6155246804...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>combineLostFrac</code> of similar type (<code>llvm::LostFrac</code>) are easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>75312e20eb...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>HasUnbound</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>17744c0772...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>upFromBoundary</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>08c9ad59d0...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>partAsInt</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>e36909f4df...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>makeNaN</code> of similar type (<code>bool</code>) are easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9e790ef121...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>readMultiplyAdd</code> of similar type (<code>const llvm::detail::IEEEFloat &amp;</code>) are easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>bb4d962509...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>convertToSignedInteger</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>78676624e...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>convertToInteger</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>a03728b4b7...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>convertFromSignedInteger</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4f83855d74...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>convertFromZeroSignedInteger</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>e530675ebd...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>roundSignAndWithExponent</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>c27b83b843...</td>
<td>.cpp</td>
<td>2 adjacent parameters for <code>convertToHexString</code> of convertible types may be easily swapped by mistake</td>
<td>experimental-cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type</td>
<td>clang-tidy</td>
<td>U</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
“That sounds like a potentially extremely helpful static analysis.”
disable bugprone-easily-swappable-parameters
clang-tidy can be a nugget sometimes x3
# REACTIONS

<table>
<thead>
<tr>
<th>Sentiment</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>64 (26 comments)</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>9 (7 comments)</td>
</tr>
</tbody>
</table>
Tool demo
Arguments

Questions
Philosophy
Arguments & Parameters
Tool demo
Arguments
Arguments

Questions
Philosophy
Arguments & Parameters & Everything else?
Introduction

- **Strong typing**
  - What? & How?
  - Negative overhead?
  - Some existing libraries
- Type migration – by hand

Overview

In detail

Tooling & “Demo”
**STRONGER** (?) **TYPES**

```cpp
int threshold(int Temperature) { return 3 * Temperature; }

int SensorTemp = /* ... */;
int T2 = threshold(SensorTemp);
print(T2);

class temperature { /* ??? */ };
/* ??? */

temperature threshold(temperature T) { return 3 * T; }

temperature Sensor = /* ... */;

temperature T2 = threshold(Sensor);
print(static_cast<int>(T2));
```
void drawBad(int width, int height);

struct Width {
    int value;

    explicit Width(int v) : value(v) {}
    explicit operator int() const {
        return value;
    }
    int operator()() const {
        return value;
    }
};

struct Height { /* Analogous... */ };
void draw(Width w, Height h) {
    int wi1 = w.value, he1 = h.value;
    int wi2 = w(), he2 = h();
    int wi3 = (int)w, he3 = static_cast<int>(h);
    int wi4{w}, he4{h};
}

// error: no implicit conversion from 'int' to 'Width'
draw(1, 2, RED);

// compile error, type mismatch
draw(Height{2}, Width{1}, RED);

draw(Width{1}, Height{2}, RED); // ✓ Works!
#include <iostream>

struct Width {
    explicit Width(int i) : Value(i) {}
    explicit operator int() const { return Value; }
    int operator[](int i) const { return Value; }
    private:
        int Value;
};

struct Height {
    explicit Height(int i) : Value(i) {}
    explicit operator int() const { return Value; }
    int operator[](int i) const { return Value; }
    private:
        int Value;
};

void drawRectangle(Width W, Height H) {
    std::cout << W() << ' ' << H() << 'n';
}

int main() {
    int Wval, Hval;
    std::cin >> Wval >> Hval;
    Width W(Wval);
    Height H(Hval);
    drawRectangle(W, H);
    return 0;
}
#include <iostream>

struct Width {
    explicit Width(int i) : Value(i) {}
    explicit operator int() const { return Value; }
    int operator[](int) const { return Value; }

    private:
    int Value;
};

struct Height {
    explicit Height(int i) : Value(i) {}
    explicit operator int() const { return Value; }
    int operator[](int) const { return Value; }

    private:
    int Value;
};

void drawRectangle(Width W, Height H) {
    std::cout << W() << ' ' << H() << 'n';
}

int main() {
    int Wval, Hval;
    std::cin >> Wval >> Hval;

    Width W(Wval);
    Height H(Hval);

    drawRectangle(W, H);

    return 0;
}
struct Width {
    int value;
    explicit Width(int v) : value(v) {}
    explicit operator int() const {
        return value;
    }
    int operator()() const {
        return value;
    }
};

struct Height { /* Analogous... */ };
struct Area { /* Analogous... */ };
Area \texttt{operator\texttt{*}}(\texttt{Width} w, \texttt{Height} h) \{ 
\texttt{return Area\{w.get\texttt{()}} * h.get\texttt{()}\}; 
\}

#include <chrono>
using namespace std::chrono;
using namespace std::literals;

// Bad: prone to bad order of arguments.
bool submit_at_1(int year, int month, int day,
                  int hour, int minute, int second);

// Bad: ”seconds” is not descriptive.
bool submit_at_2(double seconds);

// Order of arguments mixed up.
sSubmit_at_1(23, 59, 59, 2023, 1, 31);
STRONG TYPE II

// Semantically incorrect, yet compiles.
submit_at_2(getMilliseconds());

bool submit_at_good(time_point<system_clock, seconds> T) {
    auto DLDay = 2023y / January / 31;
    auto DLSecond = 24h - 1s; // = 86 399 sec
    auto AOEDeadline = zoned_time(
                        "Etc/GMT+12", DLDay + DLSecond);
    return T <= AOEDeadline.get_sys_time();
}

submit_at_good(2022); // \* compile error: no conversion. 
submit_at_good(system_clock::now());
```c
int f(int x, int y) {
    x = 512;
    y = 42;
    return x / y;
}

int main() {
    int x = 0;
    return f(x, x);
}

struct IntWrap {
    int V;
    IntWrap(int v) : V(v) {}  // V = v; return this;
    operator int() const { return V; }
}

struct Nom : IntWrap {};
struct Den : IntWrap {};

int f(Nom n, Den d) {
    n.V = 512;
    d.V = 42;
    return n.V / d.V;
}

int main() {
    Nom N();  // N(0);
    Den D();  // D(0);
    return f(N(), D);
}
```
int f(int x, int y) {
    x = 512;
    y = 42;
    return x / y;
}

int main() {
    int x = 0;
    return f(x, x);
}

struct IntWrap {
    int V;
    IntWrap(int v) : V(v) {} // Constructor
    IntWrap& operator=(int v) { V = v; return *this; } // Assignment operator
    operator int() const { return V; } // Cast operator
};

struct Nom : IntWrap {
    Nom() : IntWrap(0) {} // Constructor
};

struct Den : IntWrap {
    Den() : IntWrap(1) {} // Constructor
};

int f(Nom& n, Den& d) {
    n.V = 512;
    d.V = 42;
    return n.V / d.V;
}

int main() {
    Nom N(0); Den D(0);
    return f(N, D);
}
```c
int f(int x, int y) {
    x = 512;
    y = 42;
    return x / y;
}

int main() {
    int x = 0;
    return f(x, x);
}
```

```c
struct IntWrap
{
    int V;
    IntWrap(int v) : V(v) {} IntWrap &operator=(int v) { V = v; return *this; }
    operator int() const { return V; }
};

struct Nom : IntWrap {}
struct Den : IntWrap {}

int f(Nom n, Den d) {
    n.V = 512;
    d.V = 42;
    return n.V / d.V;
}

int main() {
    Nom N(0); Den D(0);
    return f(N, D);
}
```
#include <units/isq/si/area.h>
#include <units/isq/si/frequency.h>
#include <units/isq/si/length.h>
#include <units/isq/si/speed.h>
#include <units/isq/si/time.h>

using namespace units::isq::si::references;

// simple numeric operations
static_assert(10 * km / 2 == 5 * km);

// unit conversions
static_assert(1 * h == 3600 * s);
static_assert(1 * km + 1 * m == 1001 * m);

// dimension conversions
inline constexpr auto kmph = km / h;
static_assert(2 * kmph * (2 * h) == 4 * km);
static_assert(2 * km / (2 * kmph) == 1 * h);
static_assert(2 * m * (3 * m) == 6 * m2);
static_assert(10 * km / (5 * km) == 2);
static_assert(1000 / (1 * s) == 1 * kHz);

---

1MATEUSZ PUSZ. Implementing Physical Units Library for C++. C++Now 2019. May 9, 2019. URL: youtube.com/watch?v=wKchCktZPHU.
using namespace pssst;

struct Voltage : Linear<double, Voltage, Out> {};  
struct Current : Linear<double, Current, Out> {};  
struct Resistance : Linear<double, Resistance, Out> {};  

Voltage operator""_V(long double val) {
    return Voltage{static_cast<double>(val)};
}

Current operator/(Voltage v, Resistance r) {
    auto [vv] = v;       auto [rr] = r;
    return { vv / rr } ;
}

Resistance operator/(Voltage v, Current c) {
    auto [vv] = v;       auto [cc] = c;
    return { vv / cc } ;
}

Voltage operator*(Resistance r, Current c) {
    auto [rr] = r;       auto [cc] = c;
    return { rr * cc } ;
}

Voltage operator*(Current c, Resistance r) {
    auto [rr] = r;       auto [cc] = c;
    return { rr * cc } ;
}

auto result1 = 10_V/100_Ohm;   
ASSERT_EQUAL(0.1_A, result1);

auto result2 = 10_V/0.1_A;   
ASSERT_EQUAL(100_Ohm, result2);

---

2Peter Sommerlad. Simplest Strong Typing instead of Language Proposal (P0109).  
ESTABLISHED ALGEBRÆ

(a) absl::Duration factory functions: convert numeric values to
absl::Duration value at the given scale.
double absl1::ToDoubleMinutes(absl::Duration);
double absl1::ToDoubleSeconds(absl::Duration);
double absl1::ToDoubleMilliSeconds(absl::Duration);

(b) absl1::Duration conversion functions: convert an
absl::Duration to the indicated scale.
absl1::Time absl1::FromUnixSeconds(uint64_t);
absl1::Time absl1::FromUnixMinutes(uint64_t);

(c) absl1::Time factory functions: convert a numeric value at the
given scale since the Unix epoch to an absl1::Time representing
that time instant.
int64_t absl1::FromUnixSeconds(absl1::Time);
int64_t absl1::FromUnixMinutes(absl1::Time);
int64_t absl1::FromUnixMilliSeconds(absl1::Time);

(d) absl1::Time conversion functions: take an absl1::Time and
return a numeric value at the given scale since the Unix epoch.

TABLE I: Selections from the Absl1 Time API

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time + Duration</td>
<td>Time</td>
</tr>
<tr>
<td>Time - Duration</td>
<td>Time</td>
</tr>
<tr>
<td>Time + Time</td>
<td>Time</td>
</tr>
<tr>
<td>Time - Time</td>
<td>Undefined</td>
</tr>
<tr>
<td>Duration * Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>Duration + Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>Duration - Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>Duration + Time</td>
<td>Time</td>
</tr>
<tr>
<td>Duration - Time</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

3HYRUM K. WRIGHT. “INCREMENTAL TYPE MIGRATION USING TYPE ALGEBRA”. In: 2020 IEEE
International Conference on Software Maintenance and Evolution (ICSME). Sept. 2020,
pp. 756–765. DOI: 10.1109/ICSME46990.2020.00085. URL:
Introduction
- Strong typing
  - What? & How?
  - Negative overhead?
  - Some existing libraries
  - Type migration – by hand

Overview

In detail

Tooling & “Demo”
Assume that you have already...

- Some existing software project
- A well-designed Strong Type library
Assume that you have already...

- Some existing software project
- A well-designed Strong Type library

Let’s refactor!
int readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T; }

// ----- 8< ----- >8 ----- 

int ST = readSensor(0);
int T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);
```c
int readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T; }

// ----- 8< ----- >8 ----- 

int ST = readSensor(0);
int T2 = threshold(ST);

print(ST, T2,
     ST < T2 ? ICON_HEATING : ICON_COOLING);
```
int readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T;
}

// ----- 8< ----- >8 -----

int ST = readSensor(0);
int T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);

struct Temperature {
    int Value;
    Temperature() : Value({}) {}
    Temperature(int V) : Value(V) {}
    operator int() const { return Value; }
};
int readSensor(int ID);
void print(...);

#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T; }

// ----- 8< ----- >8 ----- 

Temperature ST = readSensor(0);
int T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);
int readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T; }

// ----- 8< ----- >8 ----- 

Temperature ST = readSensor(0);
int T2 = threshold(ST); // ← Implicit conversion.

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);
```c
int readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T; }

// ----- 8< ----- >8 -----
Temperature ST = readSensor(0);
int T2 = threshold(ST); // ← Implicit conversion?

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);
```

```c
struct Temperature {
    int Value;

    Temperature() : Value({}) {}
    explicit Temperature(int V) : Value(V) {}
    explicit operator int() const {
        return Value;
    }
};
```
int readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

int threshold(int T) { return 3 * T; }

// ----- 8< ----- >8 ----- 

Temperature ST = readSensor(0);
int T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);

struct Temperature {
    int Value;

    Temperature() : Value({}) {}
    explicit Temperature(int V) : Value(V) {}
    explicit operator int() const {
        return Value;
    }
};

error: candidate function not viable:
    no known conversion
from 'Temperature' to 'int'
Temperature readSensor(int ID);

void print(...);

#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

Temperature threshold(Temperature T) {
    return 3 * T;
}

// ----- 8< ----- >8 -----  
Temperature ST = readSensor(0);
Temperature T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);

struct Temperature {
    int Value;

    Temperature() : Value({}) {}
    explicit Temperature(int V) : Value(V) {}
    explicit operator int() const {
        return Value;
    }
};

error: invalid operands to binary expression ('int' and 'Temperature')
Temperature readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

Temperature threshold(Temperature T) {
    return 3 * T;
}
// ----- 8< ----- >8 ----- 
Temperature ST = readSensor(0);
Temperature T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);

struct Temperature {
    int Value;
    Temperature() : Value({}) {}
    explicit Temperature(int V) : Value(V) {}
    explicit operator int() const {
        return Value;
    }
};

Temperature operator*(int A,
    Temperature T) {
    return Temperature{A * static_cast<int>(T)};
}
IS THIS REALLY “STRONG TYPING”?

```c
Temperature readSensor(int ID);
void print(...);
#define ICON_COOLING (1 << 8)
#define ICON_HEATING (1 << 16)

Temperature threshold(Temperature T) {
    return 3 * T;
}
// ----- 8< ----- >8 ----- 
Temperature ST = readSensor(0);
Temperature T2 = threshold(ST);

print(ST, T2,
    ST < T2 ? ICON_HEATING : ICON_COOLING);

struct Temperature {
    int Value;
    Temperature() : Value({}) {}
    explicit Temperature(int V) : Value(V) {}
    explicit operator int() const {
        return Value;
    }
};

Temperature operator*(int A,
    Temperature T) {
    return Temperature{
        A * static_cast<int>(T)};
}
```
Anyway...

```cpp
Temperature readSensor(int ID);
void print(...);
enum Icon {
    Cooling = 1 << 8,
    Heating = 1 << 16,
};

Temperature threshold(Temperature T) {
    return 3 * T;
}

// ----- 8< ----- >8 ----- 
Temperature ST = readSensor(0);
Temperature T2 = threshold(ST);
print(ST, T2,
    ST < T2 ? Cooling : Heating);

struct Temperature {
    int Value;
    Temperature() : Value({}) {}
    explicit Temperature(int V) : Value(V) {}
    explicit operator int() const {
        return Value;
    }
};

Temperature operator *(int A, Temperature T) {
    return Temperature{
        A * static_cast<int>(T);
    }
}
```
#include <iostream>
Temperature readSensor(int ID);
enum Icon {
    Cooling = 1 << 8,
    Heating = 1 << 16,
};

Temperature threshold(Temperature T) {
    return 3 * T;
}

// ----- 8< ----- >8 ----- 
Temperature ST = readSensor(0);
Temperature T2 = threshold(ST);

screen << ST << T2 <<
(ST < T2 ? Cooling : Heating);

struct Temperature {
    int Value;
    Temperature() : Value({}) {}  
    explicit Temperature(int V);
    explicit operator int() const;
};

Temperature operator*(int A, Temperature T);
	error: invalid operands to binary expression
    (ostream' and 'Temperature')

note: (∗10^100) candidate function not viable...
Existing approaches usually make you design the new type. 

...and define a mapping to execute.

If the design does not cover, the perimeter of the migration will not compile.
Existing approaches usually make you *design* the new type
...and *define* a mapping to execute.
If the design does not cover, the perimeter of the migration will not compile.

*Problem*: Code does not compile $\implies$ no more tooling support.
THE BIG PROBLEM?

\[10^9 - 10^{11}\]
### Refactoring & Type Migration

Assume that you have already...

- Some existing software project
- A well-designed Strong Type library
- Preemptively existing mapping for type transition
Assume that you have already...

- Some existing software project
- A well-designed Strong Type Library
- Preemptive/existing mapping for type transition
Assume that you have already...

- Some existing software project
- Well-designed Strong Type Library
- Preemptive existing mapping for type transition
- Some sort of tooling?
Assume that you have already...

- Some existing software project
- A well-designed strong type library
- Preemptive/existing mapping for type transition
- Some sort of tooling?

Let’s refactor!
Introduction

Overview
- Initial taint
- Propagation
  - Round 1
  - Round 2
- Rewriting the code

In detail

Tooling & “Demo”
1 Introduction

2 Overview
   - Initial taint
   - Propagation
     - Round 1
     - Round 2
   - Rewriting the code

3 In detail

4 Tooling & “Demo”
int threshold(int T) {
    return 3 * T;
}

int ST = ...;
int T2 = threshold(ST);
**Step I: Seeding an Initial Taint**

```c
int threshold(int T)
{
    return 3 * T;
}
```

```c
fictive_type(temperature)

int ST = ...;
int T2 = threshold(ST);
```
```c
int threshold(int T) {
    return 3 * T;
}

[[ft(temperature)]]
int ST = ...
int T2 = threshold(ST);
```
g++ -c temperature.cpp

temperature.cpp: In function ‘int main()’:  
temperature.cpp:5:39: warning: ‘fictive_type’ attribute directive ignored [-Wattributes]
  5 |  [[fictive_type("temperature")]] int ST = read_sensor();
     | ^~
1 Introduction

2 Overview
   ■ Initial taint
   ■ Propagation
      Round 1
      Round 2
   ■ Rewriting the code

3 In detail

4 Tooling & “Demo”
int threshold(int T)
{
    return 3 * T;
}
[[ft(temperature)]]
    int ST = ...;
int T2 = threshold(ST);
```cpp
int threshold(int T) {
    return 3 * T;
}

[[ft(temperature)]]
int ST = ...
int T2 = threshold(ST);
```
int threshold(  
  [[ft(temperature)]] int T)  
{  
  return 3 * T;  
}  

[[ft(temperature)]]  
int ST = ...;  
int T2 = threshold(ST);
```c
int threshold(
    [[ft(temperature)]] int T)
{
    return 3 * T;
}

[[ft(temperature)]]
int ST = ...;

int T2 = threshold(ST);
```
```c
int threshold(
    [[ft(temperature)]] int T)
{
    return 3 * T;
}
[[ft(temperature)]]
int ST = ...;
int T2 = threshold(ST);
```

```
fn threshold

\[
\text{param } T
\]

\[
\text{return }
\]

\[
\text{\(3 \times T\)}
\]

\[
\text{var } ST
\]

\[
\text{var } T2
\]

\[
\text{init}
\]

\[
\text{call } \text{threshold}
\]

\[
\text{\(arg = ST\)}
\]

---


`test.cpp:1:30: note: left operand is 'int' literal, configure overload or refactor into variable`
```c
int threshold(
    [[ft(temperature)]] int T)
{
    int F = 3;
    return F * T;
}

[[ft(temperature)]]
int ST = ...
int T2 = threshold(ST);
```
/* factor * temperature = temperature */

int threshold(
    [[ft(temperature)]] int T)
{
    [[ft(factor)]] int F = 3;
    return F * T;
}

[[ft(temperature)]]
int ST = ...;
int T2 = threshold(ST);
/* factor * temperature = temperature */

```c
int threshold(
    [[ft(temperature)]] int T)
{
    [[ft(factor)]] int F = 3;
    return F * T;
}

[[ft(temperature)]]
int ST = ...;
int T2 = threshold(ST);
```
/* factor * temperature = temperature */

```
[[ft(temperature)]] int threshold( [[ft(temperature)]] int T )
{
    [[ft(factor)]] int F = 3;
    return F * T;
}

[[ft(temperature)]]
int ST = ...;
int T2 = threshold(ST);
```
/* factor * temperature = temperature */

```c
[[ft(temperature)]] int threshold([[[ft(temperature)]]] int T)
{
    [[ft(factor)]] int F = 3;
    return F * T;
}

[[ft(temperature)]]
int ST = ...;
[[ft(temperature)]]
int T2 = threshold(ST);
```
Introduction

Overview
- Initial taint
- Propagation
  - Round 1
  - Round 2
- Rewriting the code

In detail

Tooling & “Demo”
/* factor * temperature = temperature */

fn threshold
  \[
  \begin{align*}
  \text{param } T & \quad \text{var } F \\
  \text{init} & \quad \text{return} \\
  F &= 3; & F \ast T; \\
  \end{align*}
  \]

fn threshold
  \[
  \begin{align*}
  \text{init} & \quad \text{return} \\
  F &= 3; & F \ast T; \\
  \end{align*}
  \]

var ST

var T2

init

call threshold

arg = ST

[[ft(temperature)]]
int threshold(
  [[ft(temperature)]]
int T)
{
  [[ft(factor)]]
int F = 3;
return F \ast T;
}

[[ft(temperature)]]
int ST = ...;
[[ft(temperature)]]
int T2 = threshold(ST);
```cpp
int threshold(int T) {
    int F = 3;
    return F * T;
}

int ST = ...;

int T2 = threshold(ST);
```
class temperature { /* ... */ };  
class factor { /* ... */ };  

[[ft(temperature)]] int threshold(
  [[ft(temperature)]] int T)
{
  [[ft(factor)]] int F = 3;
  return F * T;
}

[[ft(temperature)]]
int ST = ...;
[[ft(temperature)]]
int T2 = threshold(ST);
```cpp
class temperature { /* ... */ };
class factor { /* ... */ };

temperature threshold(temperature T) {
    factor F{3}; // Explicit cast to strong type.
    return F * T;
}

temperature ST = ...;
temperature T2 = threshold(ST);
```
class temperature { /* ... */ };  
class factor { /* ... */ };

temperature threshold(temperature T) 
{
    factor F{3}; // Explicit cast to strong type.
    return F * T;
}

temperature ST = ...;
temperature T2 = threshold(ST);

test.cpp:11:14: error: invalid operands to binary '
' expression ('factor' and 'temperature')

    return F * T;
            ^ ~ ^ ~
class temperature { /* ... */ };  
class factor { /* ... */ };  

temperature operator *(factor F, temperature T) { /* ... */ }


temperature threshold(temperature T)  
{
    factor F{3}; // Explicit cast to strong type.
    return F * T;
}


temperature ST = ...;

temperature T2 = threshold(ST);
Introduction

Overview

In detail

- What is a “fictive type”?
  - Side note: Taint analysis
- Code generation
- Problems Design decisions
  - Ignores
  - Overloads
  - Inlines and summaries

Tooling & “Demo”
1 Introduction

2 Overview

3 In detail
   - What is a “fictive type”?
     Side note: Taint analysis
   - Code generation
   - Problems Design decisions
     Ignores
     Overloads
     Inlines and summaries

4 Tooling & “Demo”
- Transitional fuel
- In-band signal (mostly...)
- Strict “is-a”: T is-a temperature
- Transitional fuel
- In-band signal (mostly...)
- Strict “is-a”: T is-a temperature

```c
int T
```

```c
int T
```

```c
int T
```
Transitional fuel

- In-band signal (mostly...)
- Strict “is-a”: T is-a temperature

```c
[[fictive_type("temperature")]] int T
[[ft("temperature")]] int T
[[fictive_type("temperature", "velocity")]] int T
[[fictive_type("temperature"), fictive_type("velocity")]] int T
```
```c
void status_prompt_load_history()
{
    FILE *history_file, *line, *tmp;
    size_t length;

    history_file = status_prompt_find_history_file();
    if (history_file == NULL) {
        // Assuming the condition is false
        return;
    }

    log_debug("loading history from %s", history_file);

    if (fgets(line, "", history_file) == NULL) {
        // Taint originated here
        free(history_file);
        return;
    }

    // Entering loop body

    // Assuming the condition is false
    break;

    // Assuming 'length' is > 0

    // Out of bound memory access (index is tainted)
    // For more information see the checker documentation
    line[length - 1] = '\0';
    status_prompt_add_history(line);
    return;
}
```
Taint analysis

- Security-first static analysis technique
- Usually path-, flow-, and/or context sensitive
- Performed on data-flow, generally a “must all-path” problem
- Interested in the **value**
SIDE NOTE: Taint analysis

Taint analysis

- Security-first static analysis technique
- Usually path-, flow-, and/or context sensitive
- Performed on data-flow, generally a “must all-path” problem
- Interested in the value

Fictive Types

- Design-first / comprehension-first static analysis technique
- Path-, and context-insensitive, locally flow-sensitive
- Strictly a “must all-path” problem
- Interested in the type
- (Not entirely security-first, but could be adapted?)
```cpp
std::ostream& std::ostream::operator>>() (std::ostream&, [[fictive_type("maybe taint")]] char&[]);
std::FILE* std::fopen( [[fictive_type("must-not taint")]] const char*, const char*);

int main()
{
    char Buf[PATH_MAX];
    std::cin >> Buf;
    std::fopen(Buf, "r");
}
```


```cpp
#include <iostream>
#include <fstream>

int main()
{
    const char Buf[PATH_MAX];
    std::cin >> Buf;
    std::fopen(Buf, "r");
}
```

```
std::ostream& std::ostream::operator>>(
    std::ostream&, [[fictive_type("maybe taint")]] char(&)[[]]);
std::FILE* std::fopen(
    [[fictive_type("must-not taint")]] const char*, const char*);

SIDE NOTE: TAINT ANALYSIS – REVERSE PROPAGATION
```
**SIDE NOTE: Taint analysis – Forward propagation**

```cpp
std::ostream& std::ostream::operator>>(
    std::ostream&, [[fictive_type("maybe taint")]] char&[]);
std::FILE* std::fopen(
    [[fictive_type("must-not taint")]] const char*, const char*);

int main()
{
    [[fictive_type("maybe taint")]]
    char Buf[PATH_MAX];
    std::cin >> Buf;
    std::fopen(Buf, "r");
}
```
 SIDE NOTE: TAINT ANALYSIS – TRIGGER ERROR

```cpp
std::FILE* std::fopen(
    [[fictive_type("must-not taint")]]
    const char*,
    const char*);

[[fictive_type("maybe taint")]] char Buf[PATH_MAX];
std::fopen(Buf, "r");
```
int fn([[fictive_type("T")]] int P);
struct S { [[fictive_type("count")]] int Size; };
int fn([[fictive_type("T")]] int P);
struct S { [[fictive_type("count")]] int Size; };

[[ft("count")]] 5L
int int::operator+([[ft("T")]] int L, int R);
```c
int fn([[fictive_type("T")]] int P);
struct S { [[fictive_type("count")]] int Size; }

[[ft("count")]] 5L
int int::operator+([[ft("T")]] int L, int R);

OperationMap.yaml

BuiltinOperations:
  Binary:
    - Operation: "temperature - temperature"
      Result: "temperature_difference_t"
    - Operation: "time * acceleration"
      Result: "velocity"
  Commutative: true
```
Introduction

Overview

In detail

- What is a “fictive type”?  
  Side note: Taint analysis

- **Code generation**

- Problems Design decisions
  - Ignores
  - Overloads
  - Inlines and summaries

Tooling & “Demo”
StrongTypes:
- TypeName: "Distance"
  Wraps: "float"
  BinaryOperations:
    - Operation: "/"
      ArgType: "Time"
      ResType: "Speed"
    - Operation: "+"
      ArgType: "Distance"
      ResType: "Distance"
    - Operation: "+="
      ArgType: "Distance"
      ResType: "Distance"

- TypeName: "Time"
  Wraps: "float"
  BinaryOperations:
    - Operation: "+"
      ArgType: "Time"
      ResType: "Time"
    - Operation: "+="
      ArgType: "Time"
      ResType: "Time"

- TypeName: "Speed"
  Wraps: "float"

# ...
root@532a7e7c74a2:/src/sys_tests/AverageSpeed# \
/usr/bin/STGenerator .:/StrongTypes.yaml
YAML configuration satisfies all validation rules. Strong types library was generated.
Update CMakeLists.txt to add library to the project:
```
```
```cpp
#include "Speed.h"
#include "Time.h"

class Distance {
  float value;

public:
  using wrapped_type = float;

  explicit constexpr Distance(const wrapped_type& V) : value(V) {}

  explicit operator wrapped_type() const {
    return value;
  }

  constexpr wrapped_type& get() {
    return value;
  }

  constexpr wrapped_type get() const {
    return value;
  }

  inline Speed operator/(const Time& rhs) const {
    return Speed{ value / static_cast<Time::wrapped_type>(rhs) };
  }

  Distance operator+(const Distance& rhs) const {
    return Distance{ value + static_cast<wrapped_type>(rhs) };
  }

  Distance& operator+=(const Distance& rhs) {
    value += static_cast<wrapped_type>(rhs);
    return *this;
  }

};
```
1 Introduction

2 Overview

3 In detail
   - What is a “fictive type”?
     *Side note: Taint analysis*
   - Code generation
   - Problems Design decisions
     - Ignores
     - Overloads
     - Inlines and summaries

4 Tooling & “Demo”
WHERE DOES THIS ALL GO WRONG?
WHERE DOES THIS ALL GO WRONG?

Unrefactorables
WHERE DOES THIS ALL GO WRONG?

Unrefactorables

- All & every library (STL, POSIX, WinAPI, other 3\textsuperscript{rd}-party)
Where does this all go wrong?

Unrefactorables

- All & every library (STL, POSIX, WinAPI, other 3rd-party)
- C API inter-op (if custom invariants are needed)
WHERE DOES THIS ALL GO WRONG?

Unrefactorables

- All & every library (STL, POSIX, WinAPI, other 3rd-party)
- C API inter-op (if custom invariants are needed)
- “Very generic” generics
WHERE DOES THIS ALL GO WRONG?

Unrefactorables

- All & every library (STL, POSIX, WinAPI, other 3rd-party)
- C API inter-op (if custom invariants are needed)
- “Very generic” generics

How to control requirements?
WHERE DOES THIS ALL GO WRONG?

Unrefactorables

- All & every library (STL, POSIX, WinAPI, other 3rd-party)
- C API inter-op (if custom invariants are needed)
- “Very generic” generics

How to control requirements?

- (Project-)inner perimeters
WHERE DOES THIS ALL GO WRONG?

Unrefactorables

- All & every library (STL, POSIX, WinAPI, other 3rd-party)
- C API inter-op (if custom invariants are needed)
- “Very generic” generics

How to control requirements?

- (Project-)inner perimeters
- Combat over-approximation
Problem

You **can not** and **must not** rewrite external, API headers.

```cpp
std::ostream& operator<<(std::ostream&, int);
int errno;
void std::vector<T>::push_back(const T&);
```
API BOUNDARIES & CONTROLLING THE PERIMETER

Mark entity as a “black hole” in the FT-domain.

```cpp
[[ft_ignore]] std::ostream& operator<<(std::ostream&, int);
[[ft_ignore]] int errno;
[[ft_ignore]] void std::vector<T>::push_back(const T&);  // Some additional questions...
```

```cpp
[[ft("temperature")]] int T;
std::cout << T;
```

~

```cpp
temperature T;
std::cout << static_cast<int>(T);
```
API BOUNDARIES & CONTROLLING THE PERIMETER

[[][fictive_type_barrier]]

Marks an entity to be the terminus for refactoring – but allow its refactoring!

[[ft_barrier]] int PseudoSink;

[[ft("temperature")]]] int T;
[[ft_barrier]] int T2 = T;
int T3 = T2;

temperature T;
temperature T2 = T;
int T3 = static_cast<int>(T2);
Problem

[[ft_ignore]] may generate verbose and ugly casts at every call.

```cpp
std::ostream& operator<<(std::ostream&, int);
std::cout << static_cast<int>(T) << static_cast<double>(V);
```
Create an overload of the refactored function and \textit{unbox} the stronger type inside.

```cpp
[[fictive_type_overload]]

std::ostream& operator<< (std::ostream&, [[ft_overload]] int);

[[ft("temperature")]] int T;
std::cout << T;
```

```cpp
std::ostream& operator<< (std::ostream& LHS,
   const temperature& RHS) {
   return (LHS << static_cast<int>(T));
}

temperature T;
std::cout << T;
```
Problem

Several families of functions where:
- refactoring is not the right option
- ignoring and evacuating the FT-domain is not the right option

```cpp
template <typename T> T max(T&&, T&&);
int max(int, int);

[[ft("temperature")]] int T1, T2;
int Tmax = max(T1, T2);
```
Problem

T1 and T2 hit the params, they get tainted and refactored. Return type is lost.

// Where did 'int max(int, int);' go?
int max(temperature, temperature);

temperature T1, T2;
int Tmax = max(T1, T2);
Evacuating the FT-domain inserts `cast`s. Return type is lost, again...

```
[[ft_ignore]] int max(int, int);

temperature T1, T2;
int Tmax = max(static_cast<int>(T1), static_cast<int>(T2));
```
Flow fictive type through the function transparently, if possible.

```cpp
[[fictive_type_inline]]

int max(int a, int b) {
    return (a < b) ? b : a;
}
// Summary: "return_type(max) == type(a) == type(b)"
```

```cpp
int max(int, int) { /* ... */ }
```

```cpp
[[ft("temperature")]]
int T1, T2;
int Tmax = max(T1, T2);
```

```cpp
int max(int, int) { /* ... */ }
```

```cpp
temperature T1, T2;
temperature Tmax{max(
    static_cast<int>(T1),
    static_cast<int>(T2)
)};
```
1 Introduction

2 Overview

3 In detail

4 Tooling & “Demo”
   - Initial setup
   - Propagation
   - “Old” infrastructure
   - Inlining
   - Iterative driver
The Timeline

- Late 2018 - Early 2019: Research into parameter passing and unsafe interfaces
- 2020 Feb.: Foundation stone laying ceremony, first commit for Fictive Types modelling in Clang
- 2020 Jul.: In-band signalling not only read, but created
- 2021 Jul.: Operation map implemented for strengthening operators over built-ins
- **2021 Dec. 9:** C-level tool works as fix-point iteration, but does not scale
The Timeline

- Late 2018 - Early 2019: Research into parameter passing and unsafe interfaces
- 2020 Feb.: Foundation stone laying ceremony, first commit for *Fictive Types* modelling in Clang
- 2020 Jul.: In-band signalling not only read, but created
- 2021 Jul.: Operation map implemented for strengthening operators over built-ins
- **2021 Dec. 9:** C-level tool works as fix-point iteration, but does not scale
- Christmas & New Year 2021→2022: Implement custom data structure to make the tool scalable
- 2022 Mar.: Serialisation (-collect) prototype works
- 2022 Spring (semester): Work on generating the strong interface with a colleague from ELTE
- 2022 Sep.: Work on refactoring and re-implementing the operation map in the new ecosystem
FICTIVE TYPE ECOSYSTEM

Inputs

- Compilation database (CMake or CodeChecker log)
**Fictive Type ecosystem**

### Inputs
- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
Inputs

- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
- The initial taint (in-band signal!) – can be versioned!
## Fictive Type Ecosystem

### Inputs

- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
- The initial taint (in-band signal!) – can be versioned!
- Configuration files (e.g., operation map) – can be versioned!
Fictive Type ecosystem

Inputs
- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
- The initial taint (in-band signal!) – can be versioned!
- Configuration files (e.g., operation map) – can be versioned!

Tools
- clang-fictive-types [compilation db]: Overall driver
**Fictive Type ecosystem**

**Inputs**

- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
- The initial taint (in-band signal!) – can be versioned!
- Configuration files (e.g., operation map) – can be versioned!

**Tools**

- `clang-fictive-types [compilation db]`: Overall driver
- `clang-fictive-types-collect`: Pre-analysis synthesis (aka. “CTU-collect”)

...
Fictive Type ecosystem

Inputs
- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
- The initial taint (in-band signal!) – can be versioned!
- Configuration files (e.g., operation map) – can be versioned!

Tools
- clang-fictive-types [compilation db]: Overall driver
- clang-fictive-types-collect: Pre-analysis synthesis (aka. “CTU-collect”)
- clang-fictive-types-flow: Fix-point iteration
**Fictive Type Ecosystem**

**Inputs**
- Compilation database (CMake or CodeChecker log)
- Generated code must have been generated
- The initial taint (in-band signal!) – can be versioned!
- Configuration files (e.g., operation map) – can be versioned!

**Tools**
- `clang-fictive-types [compilation db]`: Overall driver
- `clang-fictive-types-collect`: Pre-analysis synthesis (aka. “CTU-collect”)
- `clang-fictive-types-flow`: Fix-point iteration
- `clang-fictive-types-diagnose`: Code generation and rewriting
Introduction

Overview

In detail

Tooling & “Demo”
- Initial setup
- Propagation
- “Old” infrastructure
- Inlining
- Iterative driver
Demo setup

header.h

```cpp
namespace lib {
    extern int DangerousGlobalVariable;
    int calc(int I);
}
```

main.cpp

```cpp
#include "header.h"

int main(int ArgC, char* ArgV[]) {
    int X = lib::calc(ArgC);
    int X2 = X + X + X;

    return X2;
}
```

lib.cpp

```cpp
#include "header.h"

namespace lib {
    int DangerousGlobalVariable = 42;
    int calc(int I) {
        DangerousGlobalVariable += I;
        return I;
    }
}
```
CodeChecker log -b "g++ -c lib.cpp; g++ -c main.cpp; g++ main.o lib.o -o prog" -o compile_commands.json

INFO 2023-04-13 14:58] - Starting build...
INFO 2023-04-13 14:58] - Build finished successfully.

fictive-types-test |
```json
File: compile_commands.json

```
CodeChecker log -b "g++ -c lib.cpp; g++ -c main.cpp; g++ main.o lib.o -o prog" -o compile_commands.json

[INFO 2023-04-13 14:58] - Starting build...

File: compile_commands.json

```json
[
  {
    "directory": "/local/repo/llvm-project/fictive-types-test",
    "command": "/usr/bin/g++ -c lib.cpp",
    "file": "lib.cpp"
  },
  {
    "directory": "/local/repo/llvm-project/fictive-types-test",
    "command": "/usr/bin/g++ -c main.cpp",
    "file": "main.cpp"
  }
]
```

./prog 1 2 3 4 5; echo $?; echo $(( 3 * 6 ))

fictive-types-test
Visit AST begin Thu Apr 13 14:59:40 2023
Visit AST end Thu Apr 13 14:59:40 2023
Writing Propagation data structure to output file `.tmp/tu-main.cpp.dat`
Write Data begin Thu Apr 13 14:59:40 2023
. ChunkFile Version 1
| -- 0x562a76bd0c40 Container '<root>' (id-len: 6) (Writable)
|   |   -- 0x562a76be18d0 Container '<meta>' (id-len: 6) (Writable)
|   |   |   -- 0x562a76c43660 Payloa 0x01 'Datafa' (id-len: 8) (Writable) WrittenBytes: 20
|   |   |   -- 0x562a76bd8160 Payloa 0x01 'MainFile' (id-len: 8) (Writable) WrittenBytes: 12
|   |   |   -- 0x562a76bd6dc0 Payloa 0x02 'DeclCla' (id-len: 11) (Writable) WrittenBytes: 28
|   |   |   -- 0x562a76bd8830 Payloa 0x02 'StmtCla' (id-len: 11) (Writable) WrittenBytes: 58
|   |   -- 0x562a76bd1630 Container '<string-tables>' (id-len: 15) (Writable)
|   |   |   -- 0x562a76bcecf70 Strings
|   |   |   |   -- 0x562a76bcecf70 Vector 0xFFFF 'Filepaths' (id-len: 9) NumElements = 1 (Writable) WrittenBytes: 56
|   |   |   |   -- 0x562a76bc1d60 Strings
|   |   |   |   |   -- 0x562a76bc1d60 Vector 0xFFFF 'Typenames' (id-len: 9) NumElements = 0 (Writable) WrittenBytes: 0
|   |   |   |   |   -- 0x562a76bc1d60 LookUpTable Size = 0
|   |   |   -- 0x562a76b5a00 Container '<node-vectors>' (id-len: 14) (Writable)
|   |   |   |   -- 0x562a76b5b20 Vector 0x04 'Decs' (id-len: 5) NumElements = 2 (Writable) WrittenBytes: 184
|   |   |   |   -- 0x562a76bcad0 Vector 0x08 'stmts' (id-len: 5) NumElements = 5 (Writable) WrittenBytes: 396
|   |   |   |   -- 0x562a76b30 Vector 0x08 'Locs&Ranges' (id-len: 16) NumElements = 1 (Writable) WrittenBytes: 28
|   |   |   |   -- 0x562a76b2e50 Payload 0x10 'Propagations' (id-len: 12) (Writable) WrittenBytes: 101
Write Data end Thu Apr 13 14:59:40 2023

================== Begin iteration 1 ===================
[T1/1] Working on lib.cpp
[T1/1] Working on main.cpp
[T1/1] No files remaining. Shutting down...
Iteration begun Thu Apr 13 14:59:40 2023
Iteration ended Thu Apr 13 14:59:40 2023
Iteration duration 0 sec

================== End iteration 1 ===================

************** At the beginning of iteration 2, no files are changed. **************

+ fictive-types-test |
```cpp
#include "header.h"

int main(int ArgC, char* ArgV[]) {
    [[clang::fictive_type("test_type")]] int X = lib::calc(ArgC);
    int X2 = X + X + X;
    return X2;
}
```
```cpp
int X2 = X + X + X;
return X2;
}
```
1 Introduction

2 Overview

3 In detail

4 Tooling & “Demo”
   - Initial setup
   - Propagation
   - “Old” infrastructure
   - Inlining
   - Iterative driver
fictive-types-test clang-fictive-types-flow --data-dir="./tmp" --jobs 1

Begin iteration 1

[T1/1] Working on lib.cpp
[T1/1] Working on main.cpp
  Error signalled when visiting.
  Error signalled when visiting.

[T1/1] No files remaining. Shutting down...

  int X2 = X + X + X;
  ^

<unknown>:0: note: begin reasoning about the left-hand side of the operator ...

/local/repo/llvm-project/fictive-types-test/main.cpp:6:14: note: carrying Fictive Type taint from referred declaration
  int X2 = X + X + X;

  [[clang::fictive_type("test_type")]] int X = lib::calc(ArgC);

<unknown>:0: note: ... end reasoning about the left-hand side of the operator
<unknown>:0: note: begin reasoning about the right-hand side of the operator ...

/local/repo/llvm-project/fictive-types-test/main.cpp:6:18: note: carrying Fictive Type taint from referred declaration
  int X2 = X + X + X;

  [[clang::fictive_type("test_type")]] int X = lib::calc(ArgC);

<unknown>:0: note: ... end reasoning about the right-hand side of the operator

Iteration begun Thu Apr 13 16:10:05 2023
Iteration ended Thu Apr 13 16:10:05 2023
Iteration duration 0 sec

End iteration 1

At the beginning of iteration 2, no files are changed. ***************
int X2 = X + X + X;

<unknown>:0: note: begin reasoning about the left-hand side of the operator ...

<unknown>:0: note: ... end reasoning about the left-hand side of the operator
<unknown>:0: note: begin reasoning about the right-hand side of the operator ...

<unknown>:0: note: ... end reasoning about the right-hand side of the operator

Iteration begun Thu Apr 13 16:14:08 2023
Iteration ended Thu Apr 13 16:14:08 2023
Iteration duration 0 sec

End iteration 1

At the beginning of iteration 2, no files are changed.

File: config.yaml

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Operations:</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

fictive-types-test cat config.yaml
Begin iteration

[T1/1] Working on lib.cpp
[T1/1] Working on main.cpp

Error signalled when visiting.

No files remaining. Shutting down...

/error: use of undeclared built-in operator 'my_other_test_t' + 'test_type'
int X2 = X + X + X;

calc as a result of built-in operator +
int X2 = X + X + X;

begin reasoning about the left-hand side of the operator ...
<unknown>:0: note: calculated as a result of built-in operator +
<unknown>:0: note: begin reasoning about the left-hand side of the operator ...
<unknown>:0: note: calculating as a result of built-in operator +
<unknown>:0: note: begin reasoning about the right-hand side of the operator ...
<unknown>:0: note: calculating as a result of built-in operator +
<unknown>:0: note: begin reasoning about the right-hand side of the operator ...

Var declaration here, tainted in the analysed source code

Var declaration here, tainted in the analysed source code

Var declaration here, tainted in the analysed source code
```
int X2 = X + X + X;
```

```
int X = lib::calc(ArgC);
```

```
int X2 = X + X + X;
```

```
int X = lib::calc(ArgC);
```

```
Iteration begun Thu Apr 13 16:38:30 2023
Iteration ended Thu Apr 13 16:38:30 2023
Iteration duration 0 sec
```

```
End iteration 1
```

```
*************** At the beginning of iteration 2, no files are changed. ***************
```

```
File: config.yaml
```

```
1
2    Operations:
3      Unary:
4          - Operation: "test_type + test_type"
5          Result: "my_other_test_t"
6          - Operation: "test_type + my_other_test_t"
7          Result: "result_t"
8          Commutative: true
```

```
```
fictive-types-test cat config.yaml

File: config.yaml

1  Operations:
  2    Unary:
  3    Binary:
  4      - Operation: "test_type + test_type"
  5        Result: "my_other_test_t"
  6      - Operation: "test_type + my_other_test_t"
  7        Result: "result_t"
  8    Commutative: true

fictive-types-test clang-fictive-types-flow --data-dir="/tmp" --jobs 1 --config-file "/config.yaml"

============= Begin iteration 1 ==============
[T1/1] Working on lib.cpp
[T1/1] Working on main.cpp
[T1/1] No files remaining. Shutting down...
/local/repo/llvm-project/fictive-types-test/main.cpp:6:9: warning: Var 'X2' declaration received Fictive Type 'result_t'
    int X2 = X + X + X;
    ~~~~~~~~~~~~~~~
/local/repo/llvm-project/fictive-types-test/main.cpp:6:20: note: calculated as a result of built-in operator +
    int X2 = X + X + X;
    ~~~~~~~~~~~~~~~~~~
<unknown>:0: remark: built-in operator + for 'my_other_test_t' and 'test_type' is defined to yield 'result_t'
<unknown>:0: note: begin reasoning about the left-hand side of the operator ...
/local/repo/llvm-project/fictive-types-test/main.cpp:6:16: note: calculated as a result of built-in operator +
    int X2 = X + X + X;
    ~~~
<unknown>:0: remark: built-in operator + for 'test_type' and 'test_type' is defined to yield 'my_other_test_t'
<unknown>:0: note: begin reasoning about the left-hand side of the operator ...
/local/repo/llvm-project/fictive-types-test/main.cpp:6:14: note: carrying Fictive Type taint from referred declaration
    int X2 = X + X + X;
    ^
<unknown>:0: note: begin reasoning about the left-hand side of the operator ...
[[clang::fictive_type("test_type")]] int X = lib::calc(ArgC);
int X2 = X + X + X;

```cpp
int X = lib::calc(Arg0);
```
1 Introduction

2 Overview

3 In detail

4 Tooling & “Demo”
   - Initial setup
   - Propagation
   - “Old” infrastructure
   - Inlining
   - Iterative driver
The Timeline

- 2020 Feb.: Foundation stone laying ceremony, first commit for *Fictive Types* modelling in Clang
- **2021 Dec. 9:** C-level tool works as fix-point iteration, but does not scale
- Christmas & New Year 2021→2022: Implement custom data structure to make the tool scalable
- 2022 Mar.: Serialisation (-collect) prototype works
#include "header.h"

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
    int X = lib::calc(ArgC);
    int X2 = X + X + X;

    return X2;
}
```cpp
#include "header.h"

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
    int X = lib::calc(ArgC);
    int X2 = X + X + X;

    return X2;
}
```

```
In file included from main.cpp:1:
./header.h:7:16: warning: parameter should be attributed with fictive type 'test_type'
    int calc(int I);
        ^
[[clang::fictive_type("test_type")]]
main.cpp:5:23: warning: passing an expression that carries the fictive type as argument
    int X = lib::calc(ArgC);
            ^~~~
main.cpp:3:51: note: expression references variable 'ArgC' that has fictive type declared
int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
        ^
```
```cpp
#include "header.h"

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
    int X = lib::calc(ArgC);
    int X2 = X + X + X;

    return X2;
}
```

```
./header.h:7:16: warning: parameter should be attributed with fictive type 'test_type'
    int calc(int I);
    ^

[[clang::fictive_type("test_type")]]

```

```
./main.cpp:5:23: error: passing an expression that carries the fictive type as argument
    int X = lib::calc(ArgC);
    ^~~

main.cpp:3:51: note: expression references variable 'ArgC' that has fictive type declared
int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
    ^
```

```
fictive-types-test clang-fictive-types-propagator -p ./main.cpp
In file included from main.cpp:1:
./header.h:7:16: warning: parameter should be attributed with fictive type 'test_type'
  int calc(int I);
    ^

[[clang::fictive_type("test_type")]]
main.cpp:5:23: note: passing an expression that carries the fictive type as argument
  int X = lib::calc(ArgC);
    ^~~~
main.cpp:3:51: note: expression references variable 'ArgC' that has fictive type declared
int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
      ^

fictive-types-test clang-fictive-types-propagator -p ./main.cpp
fictive-types-test clang-fictive-types-propagator -p ./lib.cpp
lib.cpp:9:29: error: use of undeclared built-in binary operator '<no fictive type>' + 'test_type'
        DangerousGlobalVariable += I;
        ^
lib.cpp:7:7: warning: function should be attributed with fictive type 'test_type'
  int calc(int I)
    ^

[[clang::fictive_type("test_type")]]
lib.cpp:10:12: note: returning of an expression that carries the fictive type
   return I;
     ^
lib.cpp:7:16: note: expression references variable 'I' that has fictive type declared
  int calc(int I)
    ^

In file included from lib.cpp:1:
./header.h:7:7: warning: declaration related to previous diagnostic should be attributed with fictive type 'test_type'
  int calc([[clang::fictive_type("test_type")]] int I);
    ^

[[clang::fictive_type("test_type")]]
lib.cpp:7:7: note: to keep in sync with the definition here
  int calc(int I)
    ^

fictive-types-test
```c
#pragma once

namespace lib {
    [[clang::fictive_type("global")] extern int DangerousGlobalVariable;

    int calc([[clang::fictive_type("test_type")] int I);
}
```

```
int calc(int I) ^

[[clang::fictive_type("test_type")]]
```

```
lib.cpp:10:12: note: returning of an expression that carries the fictive type
      return I;
      ^
```

```
lib.cpp:7:16: note: expression references variable 'I' that has fictive type declared
      int calc(int I)
      ^
```

In file included from lib.cpp:
./header.h:7:7: warning: declaration related to previous diagnostic should be attributed with fictive type 'test_type'
      int calc([[clang::fictive_type("test_type")] int I);
      ^

[[clang::fictive_type("test_type")]]
```

```
lib.cpp:7:7: note: to keep in sync with the definition here
      int calc(int I)
      ^
```

```
```cpp
#pragma once

namespace lib {
    [[clang::fictive_type("global")]] extern int DangerousGlobalVariable;

    int calc([[clang::fictive_type("test_type")]] int I);
}
```

```cpp
int calc(int I) {
    [[clang::fictive_type("test_type")]]
    return I;
}
```

```cpp
In file included from lib.cpp:
./header.h:7:7: warning: declaration related to previous diagnostic should be attributed with fictive type 'test_type'
    int calc([[clang::fictive_type("test_type")]] int I);
```

```cpp
./header.h:7:7: warning: function should be attributed with fictive type 'test_type'
    int calc(int I) {
```

```cpp
/lib.cpp:7:16: note: expression references variable 'I' that has fictive type declared
    int calc(int I)
```

```cpp
/lib.cpp:7:12: note: returning of an expression that carries the fictive type
    return I;
```

```cpp
/lib.cpp:7:16: note: expression references variable 'I' that has fictive type declared
    int calc(int I)
```

```
```
```
```
```
main.cpp:5:9: warning: variable should be attributed with fictive type 'test_type'
   int X = lib::calc(ArgC);
   ^
   [[clang::fictive_type("test_type")]]
main.cpp:5:13: note: variable initialised from an expression that taints with fictive type
   int X = lib::calc(ArgC);
         ^
   [[clang::fictive_type("test_type")]]
./header.h:7:44: note: the value of the called function 'calc' is declared to carry the fictive type
   [ [ clang::fictive_type("test_type") ] ] int calc([ [ clang::fictive_type("test_type") ] ] int I);
   ^
main.cpp:6:20: error: use of undeclared built-in binary operator '<no fictive type>' + 'test_type'
   int X2 = X + X + X;
       ^
   [ [ clang::fictive_type("test_type") ] ]

fictive-types-test
1 Introduction

2 Overview

3 In detail

4 Tooling & “Demo”
   - Initial setup
   - Propagation
   - “Old” infrastructure
   - Inlining
   - Iterative driver
namespace lib
{
  [[clang::fictive_type_inline]] int calc(int I)
  {
    return I;
  }
}

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
  int X = lib::calc(ArgC);
  int X2 = X + X;
  return X2;
}
```cpp
int main([[clang::fictive_type("test_type")]]) int ArgC, char* ArgV[])
{
    int X = lib::calc(ArgC);
    int X2 = X + X;
    return X2;
}
```

```
fictive-types-test clang-fictive-types-propagator -p --operation-map=./config.yaml ./main.cpp
main.cpp:3:38: remark: summary created: function 'calc' returns the fictive type of the 1st parameter
[[clang::fictive_type_inline]] int calc(int I)
    ^
main.cpp:5:12: note: returning of an expression that carries the fictive type
  return I;
  ^
main.cpp:3:47: note: expression references variable 'I' that is tainted in current execution
[[clang::fictive_type_inline]] int calc(int I)
    ^
main.cpp:11:9: warning: variable should be attributed with fictive type 'test_type'
int X = lib::calc(ArgC);
```

main.cpp:3:38: remark: summary created: function 'calc' returns the fictive type of the 1st parameter
[[clang::fictive_type::inline]] int calc(int I)

main.cpp:5:12: note: returning of an expression that carries the fictive type
  return I;

main.cpp:3:47: note: expression references variable 'I' that is tainted in current execution
[[clang::fictive_type::inline]] int calc(int I)

main.cpp:11:9: warning: variable should be attributed with fictive type 'test_type'
  int X = lib::calc(ArgC);

main.cpp:11:13: note: variable initialised from an expression that taints with fictive type
  int X = lib::calc(ArgC);

main.cpp:11:13: remark: summary applied: the called function 'calc' has been calculated to return the fictive type of the 1st argument
  int X = lib::calc(ArgC);

main.cpp:9:51: note: expression references variable 'ArgC' that has fictive type declared
int main([[clang::fictive_type::"test_type"]]) int ArgC, char* ArgV[])

main.cpp:12:16: error: use of undeclared built-in binary operator 'test_type' + 'test_type'
  int X2 = X + X;

fictive-types-test cat config.yaml

<table>
<thead>
<tr>
<th>File: config.yaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

fictive-types-test
main.cpp:3:47: note: expression references variable 'I' that is tainted in current execution
[[clang::fictive_type_inline]] int calc(int I)

main.cpp:9:5: warning: function should be attributed with fictive type 'test_type'
int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])

[[clang::fictive_type("test_type")]]
main.cpp:14:12: note: returning of an expression that carries the fictive type
    return X2;

main.cpp:12:9: note: expression references variable 'X2' that is tainted in current execution
    int X2 = X + X;

main.cpp:11:9: warning: variable should be attributed with fictive type 'test_type'
    int X = lib::calc(ArgC);

[[clang::fictive_type("test_type")]]
main.cpp:11:13: note: variable initialised from an expression that taints with fictive type
    int X = lib::calc(ArgC);

main.cpp:11:13: remark: summary applied: the called function 'calc' has been calculated to return the fictive type of the 1st argument
    int X = lib::calc(ArgC);

main.cpp:9:51: note: expression references variable 'ArgC' that has fictive type declared
int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])

main.cpp:12:9: warning: variable should be attributed with fictive type 'test_type'
    int X2 = X + X;

[[clang::fictive_type("test_type")]]
main.cpp:12:16: note: variable initialised from an expression that taints with fictive type
    int X2 = X + X;

remark: built-in operator + call for 'test_type' and 'test_type' is defined to yield 'test_type'
  fictive-types-test
1 Introduction

2 Overview

3 In detail

4 Tooling & “Demo”
   - Initial setup
   - Propagation
   - “Old” infrastructure
   - Inlining
   - Iterative driver
```cpp
namespace lib {
  int calc(int I) {
    return I;
  }
}

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[]) {
  int X = lib::calc(ArgC);
  int X2 = X + X;
  return X2;
}
```
```cpp
int X = lib::calc(ArgC);
int X2 = X + X;
return X2;
}
```

```cpp
namespace lib
{
  [[clang::fictive_type("test_type")]] int calc([[clang::fictive_type("test_type")]] int I)
  {
    return I;
  }
}

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
  [[clang::fictive_type("test_type")]] int X = lib::calc(ArgC);
  int X2 = X + X;
  return X2;
}
```
namespace lib
{
    [[clang::fictive_type_inline]] int calc(int I)
    {
        return I;
    }
}

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
    int X = lib::calc(ArgC);
    int X2 = X + X;
    return X2;
}
int X = lib::calc(ArgC);
int X2 = X + X;
return X2;
}

fictive-types-test clang-fictive-types ./compile_commands.json
Beginning next iteration...
[1/1] Running for main.cpp
/tmp/clang-fictive-types-4a4f25/compile_commands.json
/tmp/clang-fictive-types-4a4f25/main.cpp-1.yaml
Performing rewrites...
Beginning next iteration...
[1/1] Running for main.cpp
/tmp/clang-fictive-types-4a4f25/compile_commands.json
No more replacements generated.
fictive-types-test cat main.cpp

File: main.cpp

namespace lib
{
  [[clang::fictive_type_inline]] int calc(int I)
  {
    return I;
  }
}

int main([[clang::fictive_type("test_type")]] int ArgC, char* ArgV[])
{
  [[clang::fictive_type("test_type")]] int X = lib::calc(ArgC);
  int X2 = X + X;
  return X2;
}

fictive-types-test |
Formal goal achieved. Interactive discovery of interface specification is performed and possible.

Refactored code guarded by type checking.

Scalability issues in the implementation is ongoing work.
Formal goal achieved. Interactive discovery of interface specification is performed and possible.

Refactored code guarded by type checking.

Scalability issues in the implementation is ongoing work.

It’s a good question what kinds of summary are appropriate.

Supporting all features of C, and further customisation (e.g., overloads) is future work.

Supporting C++ language elements, like templates, is future work.

\[
\text{std::vector<int>} \rightarrow \text{std::vector<temperature>}
\]