Alex Skatkov

Perfect forwarding

On topic:

- Forwarding references
- Type deduction
- Reference collapsing
- Forwarding problem
- Solution

Forwarding references

Forwarding references

```
template<class T>
void foo(T&& arg) {

}
```

Forwarding references

```
template<class T>
void foo(T&& arg) {

}
```

```
auto&&
```

Forwarding references

```
template<class T>
void foo(T&& arg) {

}


auto&&
```

```
int main() {
    int var{};
    foo(100);
    foo(var);
    auto&& ref1 = 100;
    auto&& ref2 = var;
}
```

Forwarding references

```
template<class T>
void foo(T&& arg) {

}


auto&&
```

```
int main() {
    int var{};
    foo(100);   //<--
    foo(var);
    auto&& ref1 = 100;
    auto&& ref2 = var;
}
```

```
void foo(int&& arg) { }
```

Forwarding references

```
template<class T>
void foo(T&& arg) {

}


auto&&
```

```
int main() {
    int var{};
    foo(100);
    foo(var);       //<--
    auto&& ref1 = 100;
    auto&& ref2 = var;
}
```

```
void foo(int& arg) { }
```

Forwarding references

```
template<class T>
void foo(T&& arg) {

}
```

```
auto&&
```

```
int main() {
    int var{};
    foo(100);
    foo(var);
    auto&& ref1 = 100;  //<--
    auto&& ref2 = var;
}
```

```
int&& ref1 = 100;
```

Forwarding references

```
template<class T>
void foo(T&& arg) {

}
```

```
auto&&
```

```
int main() {
    int var{};
    foo(100);
    foo(var);
    auto&& ref1 = 100;
    auto&& ref2 = var;   //<--
}
```

```
int& ref2 = var;
```

Type deduction

Type deduction

```
template<class T>
void foo(T&& arg) { }
```

```
int main() {
    int var{};
    foo(5);
    foo(var);
}
```

Type deduction

```
template<class T>
void foo(T&& arg) { }
```

```
int main() {
    int var{};
    foo(5); //<--   void foo(int&& arg) { }
    foo(var);
}
```

Type deduction

```
template<class T>
void foo(T&& arg) { }
```

```
int main() {
    int var{};
    foo(5); //<--   void foo(int&& arg) { } T = int
    foo(var);
}
```

Type deduction

```
template<class T>
void foo(T&& arg) { }
```

```
int main() {
    int var{};
    foo(5);
    foo(var);   //<--   void foo(int& arg) { }
}
```

Type deduction

```
template<class T>
void foo(T&& arg) { }
```

```
int main() {
   int var{};
   foo(5);
   foo(var);   //<--   void foo(int& arg) { } T = int&
}
```

Reference collapsing

Reference collapsing

```
using T1 = int&;
```

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
```

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
```

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
```

& + & = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
```

& + & = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
```

& + & = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
```

& + & = &
& + && = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
```

& + & = &

& + && = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
using T5 = T4&;
```

& + & = &
& + && = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
using T5 = T4&;
static_assert(std::is_same_v<T5,int&>);
```

& + & = &
& + && = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
using T5 = T4&;
static_assert(std::is_same_v<T5,int&>);
```

& + & = &
& + && = &
&& + & = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
using T5 = T4&;
static_assert(std::is_same_v<T5,int&>);
using T6 = T4&&;
```

& + & = &
& + && = &
&& + & = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
using T5 = T4&;
static_assert(std::is_same_v<T5,int&>);
using T6 = T4&&;
static_assert(std::is_same_v<T6,int&&>);
```

& + & = &
& + && = &
&& + & = &

Reference collapsing

```
using T1 = int&;
using T2 = T1&;
static_assert(std::is_same_v<T2, int&>);
using T3 = T1&&;
static_assert(std::is_same_v<T3,int&>);
using T4 = int&&;
using T5 = T4&;
static_assert(std::is_same_v<T5,int&>);
using T6 = T4&&;
static_assert(std::is_same_v<T6,int&&>);
```

& + & = &
& + && = &
&& + & = &
&& + && = &&

Forwarding problem

```
struct Hard {/*impl*/};
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

cannot bind non-const lvalue reference of type ‘Hard&’ to an rvalue of type ‘Hard’

Forwarding problem

```
template<class T>
void check_and_add_to_vec(const T& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<T>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

deduced conflicting types for parameter ‘T’ (‘Hard&’ and ‘Hard’)

Forwarding problem

```
template<class T1,class T2>
void check_and_add_to_vec(T1&& val, std::vector<T2>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(var, vec);
    check_and_add_to_vec(Hard{}, vec);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(std::move(var), vec); //??
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(std::move(var), vec);
}
```

```
[T = Hard]
Hard&& val
v.push_back(val)
```

Forwarding problem

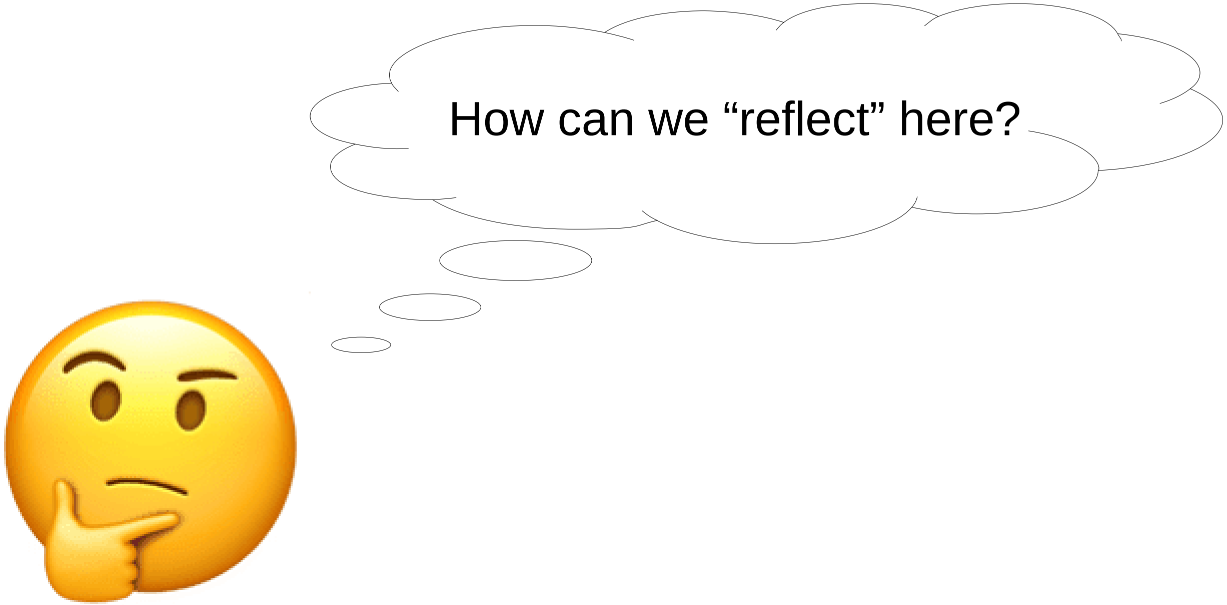
```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    //check logic...
    v.push_back(val);
}

int main() {
    Hard var{};
    std::vector<Hard> vec;
    check_and_add_to_vec(std::move(var), vec);
}
```

```
        [T = Hard]
        Hard&& val
        v.push_back(val) //Copying!
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    v.push_back(val);
}
```



How can we “reflect” here?

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    v.push_back(val);
}
```

Forwarding problem

```
template<class T>
void check_and_add_to_vec(T&& val, std::vector<std::remove_reference_t<T>>& v){
    if constexpr(std::is_lvalue_reference_v<T>) {
        v.push_back(val);
    } else {
        v.push_back(std::move(val));
    }
}
```

std::forward

std::forward

```
template<typename T>
constexpr T&&
forward(std::remove_reference_t<T>& t) noexcept {
    return static_cast<T&&>(t);
}

template<typename T>
constexpr T&&
forward(std::remove_reference_t<T>&& t) noexcept {
    static_assert(!std::is_lvalue_reference_v<T>, "template argument"
                              " substituting T is an lvalue reference type");
    return static_cast<T&&>(t);
}
```