Typing types at type level

Matthis Kruse

dasnacl
matthis-kruse

November 15, 2019

T

```
typename T
```

```cpp
template<typename T>
struct pi
{ using type = typename T::first_type; };
```

```cpp
template<typename T>
struct pi
{ using type = typename T::first_type; };

template<typename T>
using pi_t = typename pi<T>::type;
```

$$expr \ni e ::= n \mid x \mid x := e \mid e = e \mid e + e \mid e - e$$

$$expr \ni e ::= n \mid x \mid x := e \mid e = e \mid e + e \mid e - e$$

$$n \in \mathbb{N}, x \in \texttt{Vars}$$

```cpp
template<typename... Args>
struct type {};
```

```cpp
using int_t = type<int>;
using bool_t = type<bool>;
```

```cpp
template<typename... Args>
struct type {};

using fail_t = type<>;

using int_t = type<int>;
using bool_t = type<bool>;
```

```cpp
template<typename... Args> struct expr {};
```

```cpp
template<typename... Args> struct expr {};

template<int n>
using con = expr<std::integral_constant<int, n>>;
```

```
template<typename... Args> struct expr {};

template<int n>
using con = expr<std::integral_constant<int, n>>;

template<typename e1, Opr op, typename e2>
using binop = expr<std::integral_constant<Opr, op>,
                   e1, e2>;
```

```
template<typename... Args> struct expr {};

template<int n>
using con = expr<std::integral_constant<int, n>>;

template<typename e1, Opr op, typename e2>
using binop = expr<std::integral_constant<Opr, op>,
                   e1, e2>;
template<int n>
struct var_t { constexpr static int value = n; };

template<int n>
using var = expr<var_t<n>>;
```

```cpp
template<typename... Args> struct expr {};

template<int n>
using con = expr<std::integral_constant<int, n>>;

template<typename e1, Opr op, typename e2>
using binop = expr<std::integral_constant<Opr, op>,
                   e1, e2>;
template<int n>
struct var_t { constexpr static int value = n; };

template<int n>
using var = expr<var_t<n>>;

template<int x, typename e2>
using assign = expr<var<x>, e2>;
```

$$\frac{n \text{ is } \texttt{integer}}{n \to \texttt{integer}}$$

```
template<int n>
constexpr auto elab(con<n>&&) -> int_t;
```

```
template<int n>
constexpr auto elab(con<n>&&) -> int_t;

static_assert(std::is_same_v<decltype(
    elab(std::declval<con<n>>())
    ), int_t>,
  "Constants are integers.");
```

$$\frac{x \in \Gamma}{\Gamma, x \to \Gamma, \Gamma(x)}$$

```cpp
template<typename... Args>
struct list {};
```

```
template<typename... Args>
struct list {};

template<typename... Args0, typename... Args1>
constexpr auto operator+(list<Args0...>&&,
                         list<Args1...>&&)
  -> list<Args0..., Args1...>;
```

```cpp
template<typename... Args>
struct list {};

template<typename... Args0, typename... Args1>
constexpr auto operator+(list<Args0...>&&,
                         list<Args1...>&&)
  -> list<Args0..., Args1...>;

template<typename T>
constexpr auto has(T&& t, list<>&&) -> std::false_type;

template<typename T, typename... Args>
constexpr auto has(T&& t, list<Args...>&&)
  -> std::bool_constant<
       (std::is_same_v<T, Args> || ...)
     >;
```

$\Gamma(x) = \texttt{integer} \quad :\Leftrightarrow \quad \texttt{list<std::pair<var<0>, int\_t>>}$

$\Gamma(x) = \texttt{integer} \quad :\Leftrightarrow \quad \texttt{list<std::pair<var<0>, int\_t>>}$

```
template<int n>
constexpr auto elab(con<n>&&) -> int_t
```

$\Gamma(x) = \texttt{integer} \quad :\Leftrightarrow \quad \texttt{list<std::pair<var<0>, int\_t>>}$

```
template<typename Env, int n>
constexpr auto elab(Env&&, con<n>&&)
  -> std::pair<int_t, Env>
```

$$\frac{x \in \Gamma}{\Gamma, x \to \Gamma, \Gamma(x)}$$

```
template<typename Env, int x>
constexpr auto elab(Env&&, var<x>&&)
 ->
                                var<x>,
                                Env;
```

```
template<typename Env, int x>
constexpr auto elab(Env&&, var<x>&&)
 -> decltype(
       find_pair_by_component<0>(
         std::declval<var<x>>(),
         std::declval<Env>()
       )
     );
```

```
template<typename Env, int x>
constexpr auto elab(Env&&, var<x>&&)
 -> detail::elab_var_t<decltype(
      has_nth_component<0>(
        std::declval<var<x>>(),
        std::declval<Env>()
      )
    )::value, Env, var<x>>;
```

```cpp
template<bool c, typename T, typename D>
struct elab_var;

template<bool c, typename T, typename D>
using elab_var_t = typename elab_var<c, T, D>::type;
```

```
template<bool c, typename T, typename D>
struct elab_var;

template<bool c, typename T, typename D>
using elab_var_t = typename elab_var<c, T, D>::type;

template<int x, typename... Args>
struct elab_var<false, list<Args...>, var<x>>
{
  static_assert(x == 0 && false,
      "Does not support free variables.");

  using type = std::pair<fail_t, list<>>;
};
```

```
template<int x, typename... Args>
struct elab_var<true, list<Args...>, var<x>>
{
  using x_type = decltype(find_nth_component<0>(
                 std::declval<var<x>>(),
                 std::declval<list<Args...>>()));









};
```

```
template<int x, typename... Args>
struct elab_var<true, list<Args...>, var<x>>
{
  using x_type = decltype(find_nth_component<0>(
                 std::declval<var<x>>(),
                 std::declval<list<Args...>>()));

  using var_type = std::decay_t<decltype(
                 std::get<1>(std::declval<x_type>()))>;




};
```

```
template<int x, typename... Args>
struct elab_var<true, list<Args...>, var<x>>
{
  using x_type = decltype(find_nth_component<0>(
                 std::declval<var<x>>(),
                 std::declval<list<Args...>>()));

  using var_type = std::decay_t<decltype(
                 std::get<1>(std::declval<x_type>()))>;

  using type = std::conditional_t<
          !std::is_same_v<var_type, fail_t>,
           std::pair<var_type, list<Args...>>, fail_t>;
};
```

Hacking on it: `https://godbolt.org/z/vc88g4`