[GSoC] Denis first week research

Unwrap

Deprecation of `unwrapped` and `unwrapped2` because it is undecideable whether the first argument is a *callable object* or a plain value which is passed through:

```
hpx::util::unwrapped(0);

hpx::util::unwrapped([](auto) {
  // ...
})(hpx::make_ready_future(0));
```

Separation into:

- `hpx::util::unwrap`: Direct unwrap
- `hpx::util::make_unwrappable`: Deferred unwrap

I'm not sure if we really need a modernized version of `unwrapped2`, because it's currently unused (at least in the source tree).

Mapping & (async) Visitor API

Mapping

API:

```
struct future_mapping
{
    template <typename V, typename... T>
    static void of(V&& mapper, T&&...) { }
};
```

Visitor:

```
struct future_mapper
{
    template <typename T>
    using should_visit = std::true_type;

    template <typename T>
    T operator()(mocked::future<T>& f) const
    {
        // Do something on f
        return f.get();
    }
};
```

Usage:

```
future_mapping::of(future_mapper{}, make_ready_future(0), 1, 2);
```

Sync visitor

API:

```
struct future_traversal
{
    template <typename V, typename... T>
    static void of(V&& visitor, T&&...)
    {
        visitor(make_ready_future(0));
    }
};
```

Visitor:

```
struct future_visitor
{
    template <typename T>
    using should_visit = std::true_type;

    template <typename T>
    void operator()(mocked::future<T>& f) const
    {
        // Do something on f
        f.get();
    }
};
```

Usage:

```
future_traversal::of(future_visitor{}, make_ready_future(0), 1, 2);
```

Async visitor

API:

```
struct async_future_traversal
{
    template <typename V, typename... T>
    static void of(V&& visitor, T&&...)
    {
        auto current = make_ready_future(0);
        if (!visitor.touch(current)) {
            visitor.async(current, [] { /*continuation handler*/ });
            return;
        }
        // continue...
    }
};
```

Visitor:

```
struct async_future_visitor
{
    template <typename T>
    using should_visit = std::true_type;

    // Called for every future in the pack, if the method returns false,
    // the asynchronous handler is called again.
    template <typename T>
    bool operator()(future<T>& f) const
    {
        return f.is_ready(); // Call the asynchronous handler again
    }

    template <typename T, typename N>
    void operator()(future<T>& f, N&& next) const
    {
        // C++14 version may be converted to C++11
        f.then([next = std::forward<N>(next)](auto&&...) mutable {
          std::move(next)();
        });
    }

    // Called when the end is reached
    void operator()() const
    {
        // promise.resolve(...) ...
    }
};
```

Usage:

```
async_future_traversal::of(async_future_visitor{}, make_ready_future(0), 1, 2);
```

Deprecating dataflow

It was often said that `dataflow` and `when_all(...).then(...)` features the same API:

- difference `when_all` wraps into a tuple.
- `dataflow` only allocates 1 shared share
  - -> Implementation detail
- `when_all` better usable intentional because arguments are used infront
- Probably, the use is currently intentional and most users aren't aware of the state overhead of `when_all(...).then(...)`.

-> Maybe expression templates (lazy chaining of futures) through an additional template parameter in `hpx::future`, which delays the creation of the state until the future is used or type erased.

Spreads

Maybe it would be useful to also provide a `hpx::util::spreaded` function, which unwraps any future and its containing tuple:

```
hpx::util::spreaded([](std::string, std::string) {
  // ...
})(hpx::future<std::tuple<std::string, std::string>>{});

// future<T>::then
when_all(http_request("github.com"), http_request("github.com"))
    .then([](hpx::future<std::tuple<std::string, std::string>>) {
        // do something
    });

// future<T>::then_spread
when_all(http_request("github.com"), http_request("github.com"))
    .then_spread([](std::string, std::string) {
        // do something
    }); // Exceptions propagate to the next future
```