A CLOS protocol for lexical environments

Robert Strandh
Irène Durand

March, 2022

European Lisp Symposium, Porto, Portugal ELS2022

Context: The SICL project

https://github.com/robert-strandh/SICL

Several objectives:

- Create high-quality *modules* for implementors of Common Lisp systems.
- Improve existing techniques with respect to algorithms and data structures where possible.
- Improve readability and maintainability of code.
- Improve documentation.
- Ultimately, create a new implementation based on these modules.

Cleavir compiler framework

- Used by SICL.
- First pass translates a concrete syntax tree (CST) to an abstract syntax tree (AST).
- Equivalent to “minimal compilation” as defined by the Common Lisp standard.
- Special case of a “code walker”.
- Needs to maintain a *lexical compilation environment*.

Lexical compilation environment

- Reflects the nested structure of code.
- Contains information about:
  - variables (lexical, special, constant),
  - functions,
  - macros,
  - symbol macros,
  - blocks,
  - `tagbody` tags,
  - declarations that are not associated with functions or variables (in particular `optimize`).

Lexical compilation environment

- Is passed as the second argument to every macro function.
- When it is `nil`, the "null lexical environment" is designated, which is the same as the global environment.
- The Common Lisp standard does not define any operations on environment objects.
- But "Common Lisp the Language (second edition)" (CLtL2) has a section with such operators.

CLtL2 protocol

Function `variable-information`

Returns information about a name in a variable position.

- Arguments: A symbol and an optional environment.
- Values:
  1. type of binding (`:lexical`, `:special`, `:symbol-macro`, `:constant`, or `nil`).
  2. A Boolean indicating whether the binding is local.
  3. Association list of declarations that apply to the binding.

CLtL2 protocol

Function `function-information`

Returns information about a name in a function position.

- Arguments: A function name and an optional environment.
- Values:
  1. type of binding (`:function`, `:macro`, `:special-form`, or `nil`).
  2. A Boolean indicating whether the binding is local.
  3. Association list of declarations that apply to the binding.

CLtL2 protocol

Function `declaration-information`

Returns information about declarations that do not apply to a particular binding.

- Arguments: A declaration identifier and an optional environment.
- Value:
  - If the declaration identifier is `optimize`, then a list of entries of the form (*quality value*).
  - If the declaration identifier is `declaration`, then a list of declaration identifiers supplied to the `declaration` proclamation.

CLtL2 protocol

Function `augment-environment`

Given an environment, returns a new environment augmented with the given information.

Arguments:

1. An environment object.
2. Keyword arguments: `:variable`, `:symbol-macro`, `:function`, `:macro`, `:declare`.

CLtL2 protocol

Function `parse-macro`

Given a macro definition, return a macro lambda expression.

Arguments:

1. `name`. The name of the macro.
2. `lambda-list`. A macro lambda list.
3. `body`. The macro body as a list of forms, etc.
4. `env`. An optional environment. Not sure what it is used for.

CLtL2 protocol

Function `enclose`

Given a macro lambda expression and an environment, return a macro function.

Arguments:

1. `lambda-expression`. A lambda expression, possibly created by `parse-macro`.
2. `env`. An optional environment.

CLtL2 protocol

It is incomplete:

- No functionality for information about `blocks`.
- No functionality for information about `tagbodys`.
- No associated information for functions, variables, etc.

It is not possible to extend in a compatible way, because of multiple return values.

No free Common Lisp implementation we investigated (SBCL, CMUCL, ECL, CCL) uses the CLtL2 protocol for the native compiler.

Our solution

https://github.com/s-expressionists/Trucler

- Have the query functions return instances of standard classes.
- Define separate functions for each type of environment augmentation.

Our solution

Query functions:

- `describe-variable`
- `describe-function`
- `describe-block`
- `describe-tag`
- `describe-optimize`
- `describe-declarations`

Our solution

Example: `describe-function`

Parameters:

- `client`. Trucler does not specialize to this parameter. Callers should supply an instance of a standard class. Client code can define methods that specialize to their own client class(es).
- `environment`. Client code must supply an instance of a standard class, even to designate the global environment.
- `name`

Our solution

Example: `describe-function`

Returns a subclass of one of:

- `function-description`
  Subclasses:
  - `global-function-description`
    Subclass:
    - `generic-function-description`
  - `local-function-description`
- `macro-description`
  Subclasses:
  - `global-macro-description`
  - `local-macro-description`
- `special-operator-description`

Our solution

Example: `describe-function`

Accessors for `global-function-description`:

- `name`
- `type`
- `inline`
- `inline-data`
- `ignore`
- `dynamic-extent`
- `compiler-macro`

Our solution

Example:

```
(defmethod convert-cst
    (client
     cst
     (info trucler:local-macro-description)
     environment)
  (let* ((expander (trucler:expander info))
         (expanded-form
          (expand-macro expander cst environment))
         (expanded-cst
          (cst:reconstruct expanded-form cst client)))
    (setf (cst:source expanded-cst) (cst:source cst))
    (with-preserved-toplevel-ness
      (convert client expanded-cst environment))))
```

Our solution

Augmentation functions:

- `add-lexical-variable`
- `add-special-variable`
- `add-local-symbol-macro`
- `add-local-function`
- `add-local-macro`
- `add-block`
- `add-tag`

Each function returns a new environment object.

Our solution

Annotation functions:

- `add-variable-type`
- `add-variable-ignore`
- `add-variable-dynamic-extent`
- `add-function-type`
- `add-function-ignore`
- `add-function-dynamic-extent`

Each function returns a new environment object.

Our solution

Example of customization:

Client code can define a subclass of `generic-function-description` with accessors such as:

- `class-name`
- `method-class-name`

Our solution

Example of customization:

Client code can define a subclass of `variable-description` (say) `global-variable-description` if the client supports global variables.

Our solution

Advantages:

- Easier to customize and extend in compatible ways.
- Extensions can still allow for simpler code walkers to work.
- Code for different clients can co-exist in the same image.

Disadvantages:

- More consing (but we don't think consing in a code walker is a problem).
- Our functions are generic, which may cause a performance penalty in some Common Lisp implementations.

Future work

- Support more implementations (currently SBCL, CCL, and "reference")
- Improve documentation with respect to customization.
- Provide implementations of `parse-macro` and `enclose` for supported implementations.

Thank you

Questions?