Nanopass Framework Documentation

Andrew W. Keep

January 31, 2015

1 Overview

The nanopass framework provides a tool for writing compilers composed of several simple passes that operate over well-defined intermediate languages. The goal of this organization is both to simplify the understanding of each pass, because it is responsible for a single task, and to simplify the addition of new passes anywhere in the compiler.

1.1 Examples

The most complete, public example of the nanopass framework in use is in the `tests/compiler.ss` file. This is the start of a compiler implementation for a simplified subset of Scheme, used in a course on compiler implementation.

2 Defining Languages

The nanopass framework operates over a set of compiler-writer defined languages. A language definition for a simple variant of the Scheme programming language might look like:

```
(define-language L0
  (terminals
    (variable (x))
    (primitive (pr))
    (datum (d))
    (constant (c)))
  (Expr (e body)
    x
    pr
    c
    (quote d)
    (begin e* ... e)
    (if e0 e1)
    (if e0 e1 e2)
    (lambda (x* ...) body* ... body)
    (let ([x* e*] ...) body* ... body)
    (letrec ([x* e*] ...) body* ... body)
    (e0 e1 ...)))
```

The L0 language consists of a set of terminals (listed in in the **terminals** form) and a signle non-terminal Expr. The terminals of this language are variable, primitive, datum, and constant. Listed with each terminal is one or more meta-variable that can be used to indicate the terminal in a non-terminal expression. In this case variable can be represented by x, primitive can be represented by pr, datum can be represented by d, and constant can be represented by c. The nanopass framework expects the compiler writer to supply a predicate that corresponds to each terminal. The name of the predicate is derived from the name of the terminal, by adding a *?* character. For the L0 language we will need to provide variable?, primitive?, datum?, and constant? predicates. We might decided to represent variables as symbols, select a small list of primitives, represent datum as any Scheme value, and limit constants to those things that have a syntax that does not require a quote:

```
(define variable?
  (lambda (x)
    (symbol? x)))

(define primitive?
  (lambda (x)
    (memq x '(+ - * / cons car cdr pair? vector make-vector vector-length
              vector-ref vector-set! vector? string make-string
              string-length string-ref string-set! string? void))))

(define datum?
  (lambda (x)
    #t))

(define constant?
  (lambda (x)
    (or (number? x)
        (char? x)
        (string? x))))
```

The L0 language also defines one non-terminal Expr. Non-terminals start with a name (Expr) followed by a list of meta-variables ((*e body*)) and a set of productions. Each production follows one of three forms. It is either a stand alone meta-variable (in this case x, pr, or c), an S-expression that starts with a literal (in this case (**quote** d), (**begin** $e*$ ... e), (**if** *e0 e1*), etc.), or an S-expression that does not start with a literal (in this case (*e0 e1* ...). In addition to the numeric or start ($*$) suffix used in the example, a question mark (*?*) or carot ($\hat{}$) can also be used. The suffixes can also be used in combination. The suffixes do not contain any semantic value and are used simply to allow the meta-variable to be used more then once (as in the (**if** *e0 e1 e2*) form).

2.1 Extending Languages

Now that we have the L0 language we can imagine specifying a language with only the two-armed if, i.e., (**if** *e0 e1 e2*), only quoted constants, and lambda, let, and letrec bodies that contain only a single expression. One option is to simply define a whole new langauge from scratch that contains all the elements of the old language, with just the changes we want. This will work fine, but we might want to simply define what changes. The nanopass framework provides a syntax to allow a language to be defined as an extension to an already defined language:

```
(define-language L1
  (extends L0)
  (terminals
```

```
    (− (constant (c))))
  (Expr (e body)
    (− c
       (if e0 e1)
       (lambda (x* ...) body* ... body)
       (let ([x* e*] ...) body* ... body)
       (letrec ([x* e*] ...) body* ... body))
    (+ (lambda (x* ...) body)
       (let ([x* e*] ...) body)
       (letrec ([x* e*] ...) body))))
```

The newly defined language uses the **extends** form to indicate that L1 is an extension of the existing language L0. The **extends** form changes how the terminal and non-terminal forms are processed. These forms are now expected to contain a set of − forms indicating productions or terminals that should be removed or + forms indicating productions or terminals that should be added. It is also possible to add new non-terminals, by simply including a new non-terminal form with a single + form that adds all of the productions for the new non-terminal.

2.2 The Œform

Œ

The full syntax for **define-language** is as follows:

(**define-language** *language-name clause* ...)

Where *clause* is an extension clause, a entry clause, a terminals clause, or a non-terminal clause.

Extension clause. The extension clause indicates that new language is an extension of an existing language. The extension clause has the following form:

(**extends** *language-name*)

where *language-name* is the name of an already defined language. Only one extension clause can be specified in a language definition.

Entry clause. The entry clause specifies which non-terminal is the starting point for this language. This information is used when generating passes to determine which non-terminal should be expected first by the pass. This default can be overridden in a pass definition, as described in Section **??**. The entry clause has the following form:

(**entry** *non-terminal-name*)

where *non-terminal-name* corresponds to one of the non-terminals specified in this language (or the language it extends from). Only one entry clause can be specified in a language definition.

Terminals clause. The terminals clause specifies one or more terminals used by the language. For instance in the L0 example language, the terminals clause specifies four terminal types variable, primitive, datum, and constant. The terminals clause has the following form:

In a language that does not extend from a base language, the terminals clause has the following form:

(**terminals** *terminal-clause* ...)

where *terminal-clause* has one of the following forms:

(*terminal-name* (*meta-var* ...))
(=> (*terminal-name* (*meta-var* ...)) *prettifier*)
(*terminal-name* (*meta-var* ...)) => *prettifier*

Here,

- *terminal-name* is the name of the terminal and a corresponding *terminal-name?* predicate function exists to determine if a Scheme object is of this type when checking the output of a pass,
- *meta-var* is the name of a meta-variable used for referring to this terminal type in language and pass definitions,
- and *prettifier* is an expression that evaluates to a function of one argument used during when the language unparser is called in "pretty" mode to produce pretty, S-expression representation.

The final form is syntactic sugar for the form above it. When the *prettifier* is omitted, no processing will be done on the terminal when the unparser runs.

When a language derives from a base language, the terminal clause has the following form:

(**terminals** *extended-terminal-clause* ...)

where *extended-terminal-clause* has one of the following forms:

(+ *terminal-clause* ...)
(− *terminal-clause* ...)

The + form indicates terminals that should be added to the new language. The − form indicates terminals that should be removed from the list in the old language when producing the new language. Terminals not mentioned in a terminals clause will be copied into the new language, unchanged. *Note* that adding and removing *meta-var*s from a terminal currently requires removing the terminal type and re-adding it. This can be done in the same step with a terminal clause like the following:

```
(terminals
  (− (variable (x)))
  (+ (variable (x y))))
```

Non-terminals clause. The non-terminals clause specifies the valid productions in a language. Each non-terminal has a name, a set of meta-variables, and a set of productions. When the language is not extended from a base language the non-terminal clause has the following form:

```
(non-terminal-name (meta-var ...)
  production-clause
  ...)
```

where *non-terminal-name* is an identifier that names the non-terminal, *meta-var* is the name of a meta-variable used when referring to this non-terminal in a language and pass definitions, and *production-clause* has one of the following forms:

terminal-meta-var
non-terminal-meta-var
production-s-expression
(*keyword* . *production-s-expression*)

Here,

- *terminal-meta-var* is a terminal meta-variable that is a stand-alone production for this non-terminal,
- *non-terminal-meta-var* is a non-terminal meta-variable that indicates any form allowed by the specified non-terminal is also allowed by this non-terminal,
- *keyword* is an identifier that must be matched exactly when parsing an S-expression representation, language input pattern, or language output template, and
- *production-s-expression* is an S-expression that represents a pattern for language, and has the following form:

meta-variable
(**maybe** *meta-variable*)
(*production-s-expression ellipsis*)
(*production-s-expression ellipsis production-s-expression ...* . *production-s-expression*)
(*production-s-expression* . *production-s-expression*)
()

Here,

- *meta-variable* is any terminal or non-terminal meta-variable, extended with an arbitrary number of digits, followed by an arbitrary combination of $*$, *?*, or ^ characters, for example, if the meta-variable is *e* then *e1*, *e*$*$, *e?*, *e4*$*$*?* are all valid meta-variable expressions;
- (**maybe** *meta-variable*) indicates that an element in the production is either of the type of the meta-variable or bottom (represented by #f);
- and *ellipsis* is the literal ... and indicates a list of the *production-s-expression* that proceeds it is expected.

Thus, Scheme language forms such as **let** can be represented as a language production as:

(**let** ([*x*$*$ *e*$*$] ...) *body*$*$ ... *body*)

where **let** is the *keyword*, *x*$*$ is a meta-variable that indicates a list of variables, *e*$*$ & *body*$*$ are meta-variables that indicate a list of expressions, and *body* is a meta-variable that indicates a single expression. Something similar to the named-let form could also be represented as:

(**let** (**maybe** *x*) ([*x*$*$ *e*$*$] ...) *body*$*$ ... *body*)

though this would be slightly different from the normal named let form, in that the non-named form would then need an explicit #f to indicate no name was specified.

When a language extends from a base language the non-terminal clause has a slightly different form:

(*non-terminal-name* (*meta-var* ...)
  extended-production-clause
  ...)

where *extended-production-clause* has the following form:

(+ *production-clause* ...)
(− *production-clause* ...)

The + form indicates non-terminal productions that should be added to the non-terminal in the new language. The − form indicates non-terminal productions that should be removed from list of productions for this non-terminal the old language when producing the new language. Productions not mentioned in a non-terminals clause will be copied into the non-terminal in the new language, unchanged. If a non-terminal has all of its productions removed in a new language, the non-terminal will be dropped in the new language. Conversely, new non-terminals can be added by naming the new non-terminal and using the + form to specify the productions of the new non-terminal.

2.3 Products of Ø

Ø

The **define-language** form produces three user-visible objects:

- a parser (named parse-*language name*) that can be used to parse an S-expression into a record-based representation that is used internally by the nanopass framework;
- an unparser (named unparse-*language name*) that can be used to unparse a record-based representation back into an S-expression; and
- a language definition, bound to the specified *language-name*.

The language definition is used when the *language-name* is specified as the base of a new language definition and in the definition of a pass. The *language-name* can also be used with the language->s-expression to retrieve a pretty-printed version of the full language definition. This can be helpful when using extended languages, such as in the case of L1:

(language->s-expression L1)

Will return:

```
(define-language L1
  (terminals
    (variable (x))
    (primitive (pr))
    (datum (d)))
  (Expr (e body)
    (letrec ([x* e*] ...) body)
    (let ([x* e*] ...) body)
    (lambda (x* ...) body)
    x
    pr
    (quote d)
    (begin e* ... e)
    (if e0 e1 e2)
    (e0 e1 ...)))
```

3 Defining Passes

Passes are used to specify transformations over languages defined using **define-language**. Before getting into the details of defining passes, lets take a look at a simple pass to convert from L0 to L1. This pass will need to:

- Remove one armed **if**s,
- Quote constants in the language, and
- Make **begin** an explicit form in the body of the **lambda**, **let**, and **letrec** forms.

We can define a pass called make-explicit to make all of these forms explicit.

```
(define-pass make-explicit : L0 (ir) -> L1 ()
  (definitions)
  (Expr : Expr (ir) -> Expr ()
    [,x x]
    [,pr pr]
    [,c `(quote ,c)]
    [(quote ,d) `(quote ,d)]
    [(begin ,[e*] ... ,[e]) `(begin ,e* ... ,e)]
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
    [(if ,[e0] ,[e1] ,[e2]) `(if ,e0 ,e1 ,e2)]
    [(lambda (,x* ...) ,[body*] ... ,[body])
     `(lambda (,x* ...) (begin ,body* ... ,body))]
    [(let ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(let ([,x* ,e*] ...) (begin ,body* ... ,body))]
    [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(letrec ([,x* ,e*] ...) (begin ,body* ... ,body))]
    [(,[e0] ,[e1] ...) `(,e0 ,e1 ...)])
  (Expr ir))
```

The pass definition starts with a name (in this case make-explicit) and a signature. The signature starts with an input language specifier (in this case L0) along with a list of formals. In this case, we have just one formal *ir* the input-language term. The second part of the signature has an output language specifier (in this case L1) along with a list of extra return values (in this case empty).

Following the name and signature, the pass can define a set of extra **definitions** (in this case empty), a set of processors (in this case (Expr : Expr (*ir*) -> Expr () ---)), and a body expression (in this case (Expr *ir*)). Similar to the pass, a processor starts with a name (in this case Expr) and a signature. The first part of a signature is a non-terminal specifier (in this case Expr) along with the list of formals (in this case just the *ir*). Here the Expr corresponds to the one defined in the input langauge, L0. The output includes a non-terminal output specifier (in this case Expr as well) and a list of extra return expressions.

After the signature, there is a set of clauses. Each clause consists of an input pattern, an optional clause, and one or more expressions specify one or more return values based on the signature. The input pattern is derived from the S-expressions specified in the input language. Each variable in the pattern is denoted by unquote (,). When the unquote is followed by an S-expression (as is the case in of ,[*e*∗] and ,[*e*] in (**begin** ,[*e*∗] ... ,[*e*])) it indicates a cata-morphism. The cata-morphism performs automatic recursion on the subform mentioned.

Looking again at the example pass, we might notice that the (**definitions**) form is empty. When it is empty there is no need to include it. The body expression (Expr *ir*) simply calls the processor corresponding to the entry point of the input language. This can be automatically generated by **define-pass**, so we can omit this from the definition as well. Finally, we might notice that several clauses simply match the input pattern and generate an exactly matching output pattern (modulo the cata-morphisms for nested Expr forms). Since the input and output languages are defined, the **define-pass** macro can also automatically generate these clauses. So let's try again:

```
(define-pass make-explicit : L0 (ir) -> L1 ()
```

```
(Expr : Expr (ir) -> Expr ()
  [,c `(quote ,c)]
  [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
  [(lambda (,x* ...) ,[body*] ... ,[body])
   `(lambda (,x* ...) (begin ,body* ... ,body))]
  [(let ([,x* ,[e*]] ...) ,[body*] ... ,[body])
   `(let ([,x* ,e*] ...) (begin ,body* ... ,body))]
  [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
   `(letrec ([,x* ,e*] ...) (begin ,body* ... ,body))]))
```

This is a much simpler way to write the pass, though both are valid ways of writing the pass.

3.1 The -syntactic form

-

(**define-pass** *name* : *lang-specifier* (*fml* ...) -> *lang-specifier* (*extra-return-val* ...)
  definitions-clause
  processor ...
  body-expr)

```
          lang-specifier ::= language-name
                         |   (language-name non-terminal-name)
                         |   *
      definitions-clause ::= none
                         |   (definitions defn ...)
                 procesor ::= (processor-name : non-terminal-spec (fml ...) ->
                                  non-terminal-spec (extra-return-val ...)
                              definitions-clause
                              processor-clause ...)
                         |   (processor-name : * (fml ...) ->
                                  non-terminal-spec (extra-return-val ...)
                              definitions-clause
                              expr ...)
                         |   (processor-name : * (fml ...) -> * (return-val ...)
                              definitions-clause
                              body-expr ...)
                         |   (processor-name : non-terminal-spec (fml ...) -> * (return-val ...)
                              definitions-clause
                              body-processor-clause ...)
        processor-clause ::= [pattern expr expr ...]
                         |   [pattern guard expr expr ...]
                         |   [else expr expr ...]
   body-processor-clause ::= [pattern body-expr body-expr ...]
                         |   [pattern guard body-expr body-expr ...]
                         |   [else body-expr body-expr ...]
                   guard ::= (guard expr expr ...)
```

Where

- *name* is the name of the pass,

- *fml* is an identifier representing a formal argument,
- *language-name* is an identifier corresponding to a defined language,
- *non-terminal-name* is a name in that language,
- *extra-return-val* and *return-val* are Scheme expressions,
- *expr* is a Scheme expression that can contain a *quasiquote-expr*,
- *defn* is a Scheme definition, and
- *body-expr* is a Scheme expression; where
- *quasiquote-expr* is a Scheme quasiquote expression that corresponds to a form of the output non-terminal specified in a processor.

3.2 The !clause

!

The **define-pass** macro re-binds the Scheme quasiquote to use as a way to construct records in the output language. Similar to the patterns in a processor clause, the quasiquote expression can be arbitrarily nested to create language records nested within other language records. When a literal expression, such as a symbol, number, or boolean expression is one of the parts of a record, it need not be unquoted. Any literal that is not unquoted is simply put into the record as is. An unquoted expression can contain any arbitrary expression as long as the output of the expression corresponds to the type expected in the record.

When creating a quasiquoted expression, it is important to realize that while the input and output both look like arbitrary S-expressions, they are not, in fact, arbitrary S-expressions. For instance, if we look at the S-expression for a **let** in L1 we see:

(**let** ([*x*∗ *e*∗] ...) *body*)

This will create a record that expects three parts:

- a list of variables (*x*∗),
- a list of expressions (*e*∗), and
- an expression (*body*).

If this were simply an S-expression, we might imagine that we could construct an output record as:

binding*: is ([x0 e0] ... [xn en])

```
(let ([binding* (f ---)])
  `(let (,binding* ...) ,body))
```

However, this will not work because the infrastructure does not know how to destructure the *binding*∗ into its parts. Instead it is necessary to destructure the *binding*∗ list before including it in the quasiquoted expression:

binding*: is ([x0 e0] ... [xn en])

```
(let ([binding* (f ---)])
  (let ([x* (map car binding*)]
        [e* (map cadr binding*)])
    `(let ([,x* ,e*] ...) ,body)))
```

3.3 Auto-generated clauses

Within a processor a clause can be autogenerated when a matching form exists in the input-language non-terminal and the output-language non-terminal in a processor. For instance, we could eliminate the x, pr, and (**quote** d) clauses in our example pass because these forms exist in both the Expr non-terminal of the input lanugage and the Expr non-terminal of the output language. The auto-generated clauses can also process sub-forms recursively. This is why the (**begin** ,*e∗* ... ,*e*), (**if** ,*e0* ,*e1* ,*e2*), and (,*e0* ,*e1* ...) clauses can be eliminated.

The auto-generated clauses can also handle grammars with more then one non-terminal. For each sub-term in an auto-generated clause, a processor is selected. Processors are selected based on the input non-terminal and output non-terminal for the given sub-terminal. A processor with a matching input and output non-terminals is then selected. In addition to the input and output non-terminal any extra arguments required by the processor must be available, and any additional return values required by the auto-generated clause must be returned by the processor. If an appropriate processor cannot be located, a new processor will be auto-generated if possible (this is described in the next section) or an exception will be raised at compile time.

When an **else** clause is included in a processor, no clauses will be auto-generated, since it is expected that the else clause will handle all other cases.

3.4 Auto-generated processors

When a cata-morphism, auto-generated clause, or auto-generated pass body expression needs to process a sub-term, but no processor exists that can handle it, the nanopass framework might create a new processor. Processors can only be created when only a single argument in the input language and a single return value in the output language is expected, and the output-language non-terminal contains productions that match all of the input-language non-terminal. This means it is possible for the output-language non-terminal to contain extra forms, but cannot leave out any of the input terms.

An experimental feature This feature is currently implemented in the framework, but it is possible to get into trouble with this feature. In general, there are two cases where undesired clauses can be auto-generated.

1. A nested pattern in a clause matches all posible clauses, but because the pattern is nested the framework cannot determine this. In this case, the auto-generated clauses are a drag on the system, but the auto-generate processor(s) will never be called, and the code generated is effectively unreachable.
2. A cata-morphism or auto-generated clause provides a single input expression and needs a single output expression, but no processor exists that can satisfy this match. This can lead to a shadow set of processors being created and potentially undesired results being generated by a pass.

The first item is a solvable problem, though we have not yet invested the development time to fix it. The second problem is more difficult to solve, and the reason why this feature is still experimental.

3.5 Cata-morphims

Cata-morphisms are defined in patterns as an unquoted S-expression. A cata-morphism has the following syntax:

cata-morphism ::= `,[`*identifier*`...]`
| `,[`*idenitifer expr*`... ->` *identifier*`...]`
| `,[`*func-expr* `:` *identifier expr*`... ->` *identifier*`...]`
| `,[`*func-expr* `: ->` *identifer*`...]`

Where *expr* is a Scheme expression, *func-expr* is an expression that results in a function, and *identifier* is an identifier that will be bound to the input subexpression when it is on the left of the − and the return value(s) of the func when it is on the right side.

When the *func-expr* is not specified, **define-pass** attempts to find or auto-generate an appropriate processor. When the *func-expr* is the name of a processor, the input non-terminal, output non-terminal, and number of input expressions and return values will be checked to ensure they match.

If the input expressions are omitted the first input will be the sub-expression and additional values will be taken from the formals for the containing processor by name.