A Pretty Printer that Says what it Means

David Raymond Christiansen

1Supported by the Danish Advanced Technology Foundation (Højteknologifonden) grant 017-2010-3
Command: (defun addition (x y) (+ x y))
Command: (defun addition (x y) (+ x y))

ADDITION

Command:
Operation on ADDITION:

'ADDITION
(DESCRIBE 'ADDITION)
(INSPECT 'ADDITION)
ADDITION
ADDITION [will not activate]
Edit Definition (name) ADDITION (type) Function
Edit viewspecs
Show this function's argument list
Marking and yanking menu
Presentation debugging menu
System menu
Window operation menu
Command: (defun addition (x y) (+ x y))

ADDITION

Command: Show Function Arguments ADDITION
ADDITION: (X Y)
Command: □
Operation on ADDITION:

- ADDITION
- (DESCRIBE 'ADDITION)
- (INSPECT 'ADDITION)
- ADDITION [will not activate]
- Edit Definition (name) ADDITION (type) Function
- Edit viewspecs
- Show this function's argument list
- Marking and yanking menu
- Presentation debugging menu
- System menu
- Window operation menu
Command: (defun addition (x y) (+ x y))

ADDITION
Command: Show Function Arguments ADDITION
ADDITION: (X Y)
Command: (DESCRIBE 'ADDITION)
ADDITION is the function  
   #<Compiled function SI:INTERPRETER-TRAMPOLINE 21010141723>  
   #<Compiled function SI:INTERPRETER-TRAMPOLINE 21010141723> has a suffix size of 1 and a total size of 8
This function is not an internal function.
Linked references to this function are allowed.
No linked references to this function exist.
Extra info: (-2147483360 #<DTP-PACKED-INSTRUCTION-70 4013000016> 1.6560084e-24  
   (CLOS:PRINT-OBJECT SI:INTERPRETER-FUNCTION-PRINTER))

ADDITION is in the USER (really COMMON-LISP-USER) package.
ADDITION
Command:  

Dynamic Lisp Listener 3
ADDITION

Command: (defun addition (x y) (+ x y))

ADDITION

Command: Show Function Arguments ADDITION
ADDITION: (X Y)
Command: (DESCRIBE 'ADDITION)

ADDITION is the function \(<interpreted function ADDITION 104065>: (X Y)\)
  \(<interpreted function ADDITION 104065> is a lexical closure
  of the function \(<compiled function SI:INTERPRETER-TRAMPOLINE 21010141723>
  in environment (NIL SI:DIGESTED-LAMBDA
  (LAMBDA (X Y)
    (DECLARE (SYS:FUNCTION-NAME ADDITION))
    NIL
    (BLOCK ADDITION
      (+ X Y)))
  (ADDITION) 2050 655362 NIL (X Y) NIL NIL
  (BLOCK ADDITION
    (+ X Y)))

\(<compiled function SI:INTERPRETER-TRAMPOLINE 21010141723> has a suffix size of 1 and a total size
of 8

This function is not an internal function.
Linked references to this function are allowed.
No linked references to this function exist.
Extra info: \((-2147483360 #DTP-PACKED-INSTRUCTION-70 4013000016> 1.6560084e-24
  (CLOS:PRINT-OBJECT SI:INTERPRETED-FUNCTION-PRINTER))

ADDITION is in the USER (really COMMON-LISP-USER) package.

ADDITION

Command:
Presentations: a connection between screen output and the underlying objects.
The world, today

- Programming languages can no longer live apart in their own world
The world, today

- Programming languages can no longer live apart in their own world

- We must support a variety of interfaces: console, Web, editors, and more
The world, today

- Programming languages can no longer live apart in their own world

- We must support a variety of interfaces: console, Web, editors, and more

- It is useful to support clients other than development environments: IRC bots, search engines, and unknown products of ingenuity
How did we implement presentations in Idris? Can Haskell make use of the same technique?
What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types
- Primarily developed by Edwin Brady at St. Andrews, contributors around the world
- Written in Haskell
What is Idris?

- Pure functional programming language

Primarily developed by Edwin Brady at St Andrews, contributors around the world

Written in Haskell
What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types
What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types
- Primarily developed by Edwin Brady at St Andrews, contributors around the world
What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types
- Primarily developed by Edwin Brady at St Andrews, contributors around the world
- **Written in Haskell**
What is Idris like?

Demo!
How does this work?

- The pretty printing library supports **semantic annotations**
How does this work?

- The pretty printing library supports **semantic annotations**
- Annotations describe the **meaning** of sub-documents
How does this work?

- The pretty printing library supports semantic annotations
- Annotations describe the meaning of sub-documents
- Provide in-band and out-of-band communication of annotations
How does this work?

- The pretty printing library supports **semantic annotations**
- Annotations describe the **meaning** of sub-documents
- Provide in-band and out-of-band communication of annotations
- REPL commands take annotations as arguments
Concretely

Begin with a pretty-printing library:

- Hughes-Peyton Jones style
- Wadler-Leijen style
What's in a pretty printer?

- Type Doc, representing sets of strings
What's in a pretty printer?

- Type Doc, representing sets of strings
- Combinators for constructing a Doc
What's in a pretty printer?

- Type Doc, representing sets of strings
- Combinators for constructing a Doc
- Renderers that convert a Doc into a String, or to write it to a handle
1. Add a type parameter to Doc, representing the type of annotations
1. Add a type parameter to `Doc`, representing the type of annotations

2. Add the combinator
   
   `annotate :: a → Doc a → Doc a`
Make it say what it means!

1. Add a type parameter to Doc, representing the type of annotations

2. Add the combinator
   
   \[\text{annotate} :: a \to \text{Doc} \ a \to \text{Doc} \ a\]

3. Add output methods:
   
   \[\text{outputSpans} :: \text{Doc} \ a \to (\text{String}, [(\text{Int}, \text{Int}, a)])\]

   \[\text{displayDecorated} :: (a \to \text{String} \to \text{String}) \to \text{Doc} \ a \to \text{String}\]
Windows needs side effects to change console colors: ANSI codes don’t work!
Check if any elements of the first list are found in the second, using Boolean equality.

Prelude.List.hasAnyBy : (α -> α -> Bool) ->
                        List α -> List α -> Bool

Check if any elements of the first list are found in the second, using a custom comparison.

Prelude.List.hasAnyByNilFalse : (p : α -> α -> Bool) ->
                                  (l : List α) -> hasAnyBy p [] l = False

No list contains an element of the empty list by any predicate.

Prelude.List.hasAnyNilFalse : Eq α => (l : List α) ->
                              hasAny [] l = False

No list contains an element of the empty list.

Prelude.List.head : (l : List α) -> {auto ok : NonEmpty l} -> α
Get the first element of a non-empty list

Prelude.Stream.head : Stream α -> α
The first element of an infinite stream
displayDecoratedA :: (Applicative f, Monoid b)
  => (String → f b)
  → (a → f b) → (a → f b)
  → Doc a → f b
displayDecoratedA :: (Applicative f, Monoid b) => (String → f b) → (a → f b) → (Doc a → f b)

For Windows output, let f be IO and let b be ()
Because Doc is a Functor, we can transform or decorate annotations.
Because Doc is a Functor, we can transform or decorate annotations. Uses:

- Add additional type information without imposing dependencies on pretty-printer
- Convert annotations to the IDE protocol
- Type check terms that occur inside of docstrings
Implementations

annotated-wl-pprint
Idris’s pretty-printing library — a Wadler-Leijen derivative

pretty
Trevor Elliott at Galois implemented annotations for it in 2014
wl-pprint-extras
A free monad based on wl-pprint where effects can be embedded in documents
When pretty printers say what they mean, listeners don’t need to guess.
Thanks for listening!

- When pretty printers say what they mean, listeners don’t need to guess (or, even worse — parse)
Thanks for listening!

- When pretty printers say what they mean, listeners don’t need to guess (or, even worse – parse)

- Editor commands can get ahold of references directly
Thanks for listening!

➤ When pretty printers say what they mean, listeners don’t need to guess (or, even worse – parse)

➤ Editor commands can get ahold of references directly

➤ Can be implemented incrementally:
  type Doc = Annotated.Doc ()
When pretty printers say what they mean, listeners don’t need to guess (or, even worse — parse)

Editor commands can get ahold of references directly

Can be implemented incrementally:
\[
\text{type Doc = Annotated.Doc ()}
\]

Let’s take good ideas from Lisp and Smalltalk UIs, and make them \text{even better} with types!