Functional un|unparsing

Kenichi Asai (Ochanomizu) Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers)

The problem

```
printf("%d-th character after %c is %c", 5, 'a', 'f');
```

```
5-th character after a is f
```

```
scanf("%d-th character after %c is %c", &i, &c1, &c2);
```

Number and types of arguments depend on format descriptor.

Do we need dependent types?

Danvy (1998): `printf` in mere Hindley-Milner.

Today: derive `printf` and `scanf`.

The problem

```
printf("%d-th character after %c is %c", 5, 'a', 'f');

5-th character after a is f

scanf("%d-th character after %c is %c", &i, &c1, &c2);
```

Number and types of arguments depend on format descriptor.

Do we need dependent types?

Danvy (1998): `printf` in mere Hindley-Milner.

Today: derive `printf` and `scanf`.

The problem

```
printf("%d-th character after %c is %c", 5, 'a', 'f');

5-th character after a is f

scanf("%d-th character after %c is %c", &i, &c1, &c2);
```

Number and types of arguments depend on format descriptor.

Do we need dependent types?

Danvy (1998): `printf` in mere Hindley-Milner.

Today: derive `printf` and `scanf`.

The problem

```
printf("%d-th character after %c is %c", 5, 'a', 'f');

5-th character after a is f

scanf("%d-th character after %c is %c", &i, &c1, &c2);
```

Number and types of arguments depend on format descriptor.

Do we need dependent types?

Danvy (1998): `printf` in mere Hindley-Milner.

Today: derive `printf` and `scanf`.

specification

$\downdownarrows$

implementation

`scanf` `printf`

What is a spec?

> "A specification is a set of sentences in some logical language. The names of the functions, predicates, and procedures which the specification is intended to specify appear as nonlogical symbols in these sentences."
>
> —"Specifications, models, and implementations of data abstractions" (Wand 1982)

Our nonlogical symbols: `printf`, `scanf`, sequence constructors.

What is a spec?

> "A specification is a set of sentences in some logical language. The names of the functions, predicates, and procedures which the specification is intended to specify appear as nonlogical symbols in these sentences."
>
> —"Specifications, models, and implementations of data abstractions" (Wand 1982)

Our nonlogical symbols: `printf`, `scanf`, sequence constructors.

```
"%d-th character after %c is %c"
```

```
consD int (consD (lit " -th character after ")
 (consD char (consD (lit " is ") (consD char nilD))))
```

```
[int; lit "-th character after ";
 char; lit " is "; char]
```
$_D$

Specification of `printf`

```
printf [int; lit "-th character after ";
        char; lit " is "; char]D
       5 'a' 'f'
 = "5-th character after a is f"
```

Specification of `printf`

```
printf [int; lit "-th character after ";
         char; lit " is "; char]_D
        [5; (); 'a'; (); 'f']_A
 = ["5"; "-th character after "; "a"; " is "; "f"]_S
```

Specification of printf

```
printf [int; lit "-th character after ";
        char; lit " is "; char]_D
       [5; (); 'a'; (); 'f']_A
 = ["5"; "-th character after "; "a"; " is "; "f"]_S
```

```
printf nilD nilA = nilS
printf (consD (lit str) ds) (consA () xs)
 = consS str (printf ds xs)
printf (consD char ds) (consA c xs)
 = consS (string_of_char c) (printf ds xs)
printf (consD int ds) (consA i xs)
 = consS (string_of_int i) (printf ds xs)
```

Specification of printf

```
printf [int; lit "-th character after ";
        char; lit " is "; char]_D
       [5; (); 'a'; (); 'f']_A
 = ["5"; "-th character after "; "a"; " is "; "f"]_S
```

```
printf nilD nilA = nilS
printf (consD (lit str) ds) (consA () xs)
 = consS str (printf ds xs)
printf (consD char ds) (consA c xs)
 = consS (string_of_char c) (printf ds xs)
printf (consD int ds) (consA i xs)
 = consS (string_of_int i) (printf ds xs)
```

Specification of printf

```
printf [int; lit "-th character after ";
        char; lit " is "; char]_D
       [5; (); 'a'; (); 'f']_A
 = ["5"; "-th character after "; "a"; " is "; "f"]_S
```

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)
```

```
lit str () = str
char    c  = string_of_char c
int     i  = string_of_int i
```

The Interpreter Recipe

1. Look at a piece of data.
2. Decide what kind of data it represents.
3. Extract the components of the datum and do the right thing with them.

Specification of `scanf`

```
scanf [int; lit "-th character after ";
       char; lit " is "; char]_D
      "5-th character after a is f"
 = fun f -> f 5 'a' 'f'
```

Specification of `scanf`

```
scanf [int; lit "-th character after ";
       char; lit " is "; char]_D
      ["5"; "-th character after "; "a"; " is "; "f"]_S
 = [5; (); 'a'; (); 'f']_A
```

Specification of scanf

```
scanf [int; lit "-th character after ";
       char; lit " is "; char]_D
      ["5"; "-th character after "; "a"; " is "; "f"]_S
 = [5; (); 'a'; (); 'f']_A
```

```
scanf nilD nilS = nilA
scanf (consD (lit str) ds) (consS s ss)
 = consA (assert (str = s)) (scanf ds ss)
scanf (consD char ds) (consS s ss)
 = consA (char_of_string s) (scanf ds ss)
scanf (consD int ds) (consS s ss)
 = consA (int_of_string s) (scanf ds ss)
```

Specification of `scanf`

```
scanf [int; lit "-th character after ";
       char; lit " is "; char]_D
      ["5"; "-th character after "; "a"; " is "; "f"]_S
 = [5; (); 'a'; (); 'f']_A
```

```
scanf nilD nilS = nilA
scanf (consD (lit str) ds) (consS s ss)
 = consA (assert (str = s)) (scanf ds ss)
scanf (consD char ds) (consS s ss)
 = consA (char_of_string s) (scanf ds ss)
scanf (consD int ds) (consS s ss)
 = consA (int_of_string s) (scanf ds ss)
```

Specification of `scanf`

```
scanf [int; lit "-th character after ";
       char; lit " is "; char]_D
      ["5"; "-th character after "; "a"; " is "; "f"]_S
 = [5; (); 'a'; (); 'f']_A
```

```
scanf nilD nilS = nilA
scanf (consD d ds) (consS s ss)
 = consS (d s) (scanf ds ss)
```

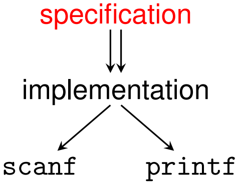
```
lit str s = assert (str = s)
char    s = char_of_string s
int     s = int_of_string s
```

Specification of `printf` and `scanf`

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)

scanf nilD nilS = nilA
scanf (consD d ds) (consS s ss)
 = consS (d s) (scanf ds ss)
```

Both just `zipWith id`!



On to implementation

Recurring idea:
fuse format descriptors with their contexts of use.
(inline; specialize)

"By considering continuations, local transformation strategies can take advantage of global knowledge."

—"Continuation-based program transformation strategies"

(Wand 1980)

$$\begin{array}{ccc} & \text{specification} & \\ & \Downarrow & \\ & \text{implementation} & \\ \swarrow & & \searrow \\ \texttt{scanf} & & \texttt{printf} \end{array}$$

Uniform implementation: deforesting format descriptors

Both `printf` and `scanf` are just `zipWith id`.

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)
```

Uniform implementation: deforesting format descriptors

Both `printf` and `scanf` are just `zipWith id`.

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)
```

It's a compositional interpreter—matching definition of a fold:

```
fold z g nil          = z
fold z g (cons x xs) = g x (fold z g xs)
```

Hence, `printf` is a fold:

```
printf = fold z g  where
    z      nilA          = nilS
    g d ds (consA x xs) = consS (d x) (ds xs)
```

Uniform implementation: deforesting format descriptors

Both `printf` and `scanf` are just `zipWith id`.

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)
```

It's a compositional interpreter—matching definition of a fold:

```
fold z g nil          = z
fold z g (cons x xs) = g x (fold z g xs)
```

Hence, `printf` is a fold, and the descriptor can be deforested:

```
printf = id
nilD       nilA          = nilS
consD d ds (consA x xs) = consS (d x) (ds xs)
```

Uniform implementation: deforesting format descriptors

Both `printf` and `scanf` are just `zipWith id`.

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)
```

It's a compositional interpreter—matching definition of a fold:

```
fold z g nil          = z
fold z g (cons x xs) = g x (fold z g xs)
```

Hence, `printf` is a fold, and the descriptor can be deforested:

```
printf = id
nilD        ()       = ()
consD d ds (x, xs) = (d x, ds xs)
```

Choose tuple representation.

Uniform implementation: deforesting format descriptors

Both `printf` and `scanf` are just `zipWith id`.

```
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
 = consS (d x) (printf ds xs)
```

It's a compositional interpreter—matching definition of a fold:

```
fold z g nil          = z
fold z g (cons x xs) = g x (fold z g xs)
```

Hence, `printf` is a fold, and the descriptor can be deforested:

```
printf = id    scanf = id
nilD       ()      = ()
consD d ds (x, xs) = (d x, ds xs)
```

Choose tuple representation. Same with `scanf`.

Not quite the standard `scanf`

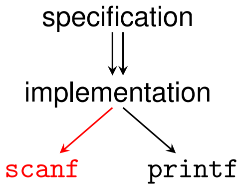
We have:

```
scanf [int; lit "-th character after "; char; lit " is ";
      ("5",("-th character after ",("a",(" is ",("f",()))
 = (5,((),('a',((),('f',())))))
```

We want:

```
scanf [int; lit "-th character after "; char; lit " is ";
      "5-th character after a is f"
 = fun f -> f 5 'a' 'f'
```

Fix: fuse primitive descriptors with `consD`.



On to the standard `scanf`

```
nilD       ()      = nilA
consD d ds (s,ss) = consA (d s) (ds ss)
```

Fuse each primitive descriptor with `consD`.

```
lit str s = assert (str = s)
char    s = char_of_string s
int     s = int_of_string s
```

On to the standard `scanf`

```
nilD        ()      = nilA
consD d = d
```

Fuse each primitive descriptor with `consD`.

```
lit str ds (s,ss) = consA (assert (str = s)) (ds ss)
char    ds (s,ss) = consA (char_of_string s) (ds ss)
int     ds (s,ss) = consA (int_of_string s ) (ds ss)
```

Primitive descriptors can consume and produce different amounts.

On to the standard `scanf`

```
nilD       ()      = nilA
consD d = d
```

Fuse each primitive descriptor with `consD`.

```
lit str ds (s,ss) = consA (assert (str = s)) (ds ss)
char    ds (s,ss) = consA (char_of_string s) (ds ss)
int     ds (s,ss) = consA (int_of_string s ) (ds ss)
```

Primitive descriptors can consume and produce different amounts.

```
char ds inp
 = if String.length inp > 0
   then consA (inp.[0])
          (ds (String.sub inp 1 (String.length inp - 1)))
   else failwith "scanf char"
```

On to the standard `scanf`

```
nilD        ()      = nilA
consD d = d
```

Fuse each primitive descriptor with `consD`.

```
lit str ds (s,ss) = consA (assert (str = s)) (ds ss)
char    ds (s,ss) = consA (char_of_string s) (ds ss)
int     ds (s,ss) = consA (int_of_string s ) (ds ss)
```

Primitive descriptors can consume and produce different amounts.

```
char ds inp
 = if String.length inp > 0
   then consA (inp.[0])
          (ds (String.sub inp 1 (String.length inp - 1)))
   else failwith "scanf char"
```

On to the standard scanf

```
nilD       ()     = nilA
consD d = d
```

Fuse each primitive descriptor with consD.

```
lit str ds (s,ss) = consA (assert (str = s)) (ds ss)
char    ds (s,ss) = consA (char_of_string s) (ds ss)
int     ds (s,ss) = consA (int_of_string s ) (ds ss)
```

Primitive descriptors can consume and produce different amounts.

```
lit str ds inp
 = if String.length str <= String.length inp &&
      str = String.sub inp 0 (String.length str)
   then ds (String.sub inp (String.length str)
                (String.length inp - String.length str))
   else failwith "scanf lit"
```

On to the standard scanf

```
nilD       ()     = nilA
consD d = d
```

Fuse each primitive descriptor with consD.

```
lit str ds (s,ss) = consA (assert (str = s)) (ds ss)
char    ds (s,ss) = consA (char_of_string s) (ds ss)
int     ds (s,ss) = consA (int_of_string s ) (ds ss)
```

Primitive descriptors can consume and produce different amounts.

```
lit str ds inp
 = if String.length str <= String.length inp &&
      str = String.sub inp 0 (String.length str)
   then ds (String.sub inp (String.length str)
               (String.length inp - String.length str))
   else failwith "scanf lit"
```

On to the standard `scanf`

```
nilD       ""     = nilA
consD d = d
```

Fuse each primitive descriptor with `consD`.

```
lit str ds (s,ss) = consA (assert (str = s)) (ds ss)
char    ds (s,ss) = consA (char_of_string s) (ds ss)
int     ds (s,ss) = consA (int_of_string s ) (ds ss)
```

Primitive descriptors can consume and produce different amounts.

Finally, Church-encode parsing results.

```
let nilA       = fun f -> f
let consA x xs = fun f -> xs (f x)
```

Done!

Not quite the standard `printf`

We have:

```
printf [int; lit "-th character after "; char; lit " is ";
       (5,((),('a',((),('f',())))))
 = ("5",("-th character after ",("a",(" is ",("f",()))))
```

We want:

```
printf [int; lit "-th character after "; char; lit " is ";
       5 'a' 'f'
 = "5-th character after a is f"
```

Fix: fuse *descriptors* with `consD`
(i.e., transform them to CPS).

$$\begin{tikzcd} & \text{specification} \arrow[d, shift left] \arrow[d, shift right] & \\ & \text{implementation} \arrow[dl] \arrow[dr] & \\ \texttt{scanf} & & \texttt{printf} \end{tikzcd}$$

On to the standard `printf`

Begin by symmetry with `scanf`:

```
printf ds = ds          nilD = ""          consD d = d
```

Input: nested tuple without `()`. Output: single string.

```
lit str ds (  xs) = str                ^ ds xs
char    ds (c,xs) = string_of_char c ^ ds xs
int     ds (i,xs) = string_of_int i  ^ ds xs
```

On to the standard `printf`

Begin by symmetry with `scanf`:

```
printf ds = ds          nilD = ""          consD d = d
```

Input: nested tuple without `()`. Output: single string.

```
lit str ds (  xs) = str                ^ ds xs
char    ds (c,xs) = string_of_char c ^ ds xs
int     ds (i,xs) = string_of_int i  ^ ds xs
```

If only we had `= (.....................) xs`
then we could just curry and eta-reduce.

On to the standard `printf`

Begin by symmetry with `scanf`:

```
printf ds = ds id     nilD k = k ""     consD d = d
```

Input: nested tuple without `()`. Output: single string.

```
lit str ds (  xs) = str                ^ ds xs
char    ds (c,xs) = string_of_char c ^ ds xs
int     ds (i,xs) = string_of_int i  ^ ds xs
```

Pass continuation to `ds`.

```
lit str ds k (  xs)
 = ds (fun s -> k (str               ^ s)) xs
char    ds k (c,xs)
 = ds (fun s -> k (string_of_char c ^ s)) xs
int     ds k (i,xs)
 = ds (fun s -> k (string_of_int i  ^ s)) xs
```

On to the standard `printf`

Begin by symmetry with `scanf`:

```
printf ds = ds id     nilD k = k ""     consD d = d
```

Input: nested tuple without `()`. Output: single string.

```
lit str ds (  xs) = str                ^ ds xs
char    ds (c,xs) = string_of_char c ^ ds xs
int     ds (i,xs) = string_of_int i  ^ ds xs
```

Pass continuation to `ds`, then curry and eta-reduce.

```
lit str ds k    xs
 = ds (fun s -> k (str              ^ s)) xs
char    ds k  c xs
 = ds (fun s -> k (string_of_char c ^ s)) xs
int     ds k  i xs
 = ds (fun s -> k (string_of_int i  ^ s)) xs
```

On to the standard `printf`

Begin by symmetry with `scanf`:

```
printf ds = ds id      nilD k = k ""     consD d = d
```

Input: nested tuple without `()`. Output: single string.

```
lit str ds (  xs) = str                ^ ds xs
char    ds (c,xs) = string_of_char c ^ ds xs
int     ds (i,xs) = string_of_int i  ^ ds xs
```

Pass continuation to `ds`, then curry and eta-reduce. Done!

```
lit str ds k
 = ds (fun s -> k (str              ^ s))
char    ds k  c
 = ds (fun s -> k (string_of_char c ^ s))
int     ds k  i
 = ds (fun s -> k (string_of_int i  ^ s))
```

Representing control

Continuation-passing style:

```
printf ds = ds id    consD d = d    nilD k = k ""
lit str ds k   = ds (fun s -> k (str              ^ s))
char    ds k c = ds (fun s -> k (string_of_char c ^ s))
int     ds k i = ds (fun s -> k (string_of_int i  ^ s))
```

A chain of closures builds up.

> "The solution is a more abstract view of the domain of continuations. What we need is an abstract algebra for modeling the rest of a computation and its operations."
>
> —"Abstract continuations" (Felleisen, Wand, Friedman & Duba 1988)

Representing control

Continuation-passing style:

```
printf ds = ds id     consD d = d     nilD k = k ""
lit str ds k   = ds (fun s -> k (str                ^ s))
char    ds k c = ds (fun s -> k (string_of_char c ^ s))
int     ds k i = ds (fun s -> k (string_of_int i  ^ s))
```

“Data-structure continuations”:

```
printf ds = ds ""     consD d = d     nilD k = k
lit str ds k   = ds (k ^ str              )
char    ds k c = ds (k ^ string_of_char c)
int     ds k i = ds (k ^ string_of_int i )
```

See paper for direct style: `consD` becomes just `^`

A new solution: `reset (fun () -> printf [...]`$_D$ `5 'a' 'f')`

Summary

"Though this be madness, yet there is method in 't."

—*Hamlet* (Shakespeare)

Principles established by Mitch are now clichés.
We use them to derive `printf` and `scanf`.

Thanks! To more decades to come.