Functional un|unparsing

Kenichi Asai (Ochanomizu)  Oleg Kiselyov (FNMOOC)
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?

Today: derive printf and scanf.
The problem

\[
\text{printf("%d-th character after %c is %c", 5, 'a', 'f');}
\]

5-th character after a is f

\[
\text{scanf("%d-th character after %c is %c", \&i, \&c1, \&c2);}
\]

Number and types of arguments depend on format descriptor.
Do we need dependent types?

**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?


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?


Today: derive printf and scanf.

specification

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"

\[
\text{consD} \text{ 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 \texttt{printf}

\begin{verbatim}
printf [int; lit "-th character after ";
    char; lit " is "; char]_D
  5 'a' 'f'
= "5-th character after a is f"
\end{verbatim}
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`

\[
\begin{align*}
\text{printf } [\text{int}; \text{lit } "-th character after "; \\
\text{char}; \text{lit } " is "; \text{char}]_D \\
[5; (); \text{'}a\text{'}; (); \text{'}f\text{'}]_A \\
= ["5"; "-th character after "; "a"; " is "; "f"]_S
\end{align*}
\]

\[
\begin{align*}
\text{printf } \text{nil}_D \text{nil}_A &= \text{nil}_S \\
\text{printf } (\text{cons}_D (\text{lit str}) \text{ ds}) (\text{cons}_A () \text{ xs}) \\
&= \text{cons}_S \text{ str } (\text{printf } \text{ ds } \text{ xs}) \\
\text{printf } (\text{cons}_D \text{ char } \text{ ds}) (\text{cons}_A \text{ c } \text{ xs}) \\
&= \text{cons}_S (\text{string_of_char c}) (\text{printf } \text{ ds } \text{ xs}) \\
\text{printf } (\text{cons}_D \text{ int } \text{ ds}) (\text{cons}_A \text{ i } \text{ xs}) \\
&= \text{cons}_S (\text{string_of_int i}) (\text{printf } \text{ ds } \text{ xs})
\end{align*}
\]
Specification of printf

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

printf \text{nilD nilA} = \text{nilS}
printf (\text{consD d ds}) (\text{consA x xs})
= \text{consS} (d \ x) (printf \text{ds xs})

\text{lit \ str} () = \str
\text{char \ c} = \text{string_of_char \ c}
\text{int \ i} = \text{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`

```plaintext
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

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

scanf nil\_D nil\_S = nil\_A
scanf (cons\_D (lit str) ds) (cons\_S s ss)
  = cons\_A (assert (str = s)) (scanf ds ss)
scanf (cons\_D char ds) (cons\_S s ss)
  = cons\_A (char_of_string s) (scanf ds ss)
scanf (cons\_D int ds) (cons\_S s ss)
  = cons\_A (int_of_string s) (scanf ds ss)
```
Specification of scanf

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

\text{scanf} \ \text{nil}_D \ \text{nil}_S = \ \text{nil}_A

\text{scanf} \ (\text{cons}_D \ (\text{lit} \ \text{str}) \ ds) \ (\text{cons}_S \ s \ ss)
\]
\[= \ \text{cons}_A \ (\text{assert} \ (\text{str} = s)) \ (\text{scanf} \ ds \ ss)\]
\text{scanf} \ (\text{cons}_D \ \text{char} \ ds) \ (\text{cons}_S \ s \ ss)
\]
\[= \ \text{cons}_A \ (\text{char\_of\_string} \ s) \ (\text{scanf} \ ds \ ss)\]
\text{scanf} \ (\text{cons}_D \ \text{int} \ ds) \ (\text{cons}_S \ s \ ss)
\]
\[= \ \text{cons}_A \ (\text{int\_of\_string} \ s) \ (\text{scanf} \ ds \ ss)\]
Specification of `scanf`

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

\[
\text{scanf \ nilD \ nilS} = \text{nilA}
\text{scanf \ (consD \ d \ ds) \ (consS \ s \ ss)}
= \text{consS} \ (d \ s) \ (\text{scanf} \ ds \ ss)
\]

\[
\text{lit \ str \ s} = \text{assert} \ (\text{str} = s)
\text{char \ s} = \text{char_of_string} \ s
\text{int \ s} = \text{int_of_string} \ s
\]
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)
Uniform implementation: deforesting format descriptors

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

\[
\begin{align*}
\text{printf } \text{nilD nilA} &= \text{nilS} \\
\text{printf } (\text{consD } d \text{ ds}) (\text{consA } x \text{ xs}) &= \text{consS} (d \ x) (\text{printf } \text{ds} \text{ xs})
\end{align*}
\]
Uniform implementation: deforesting format descriptors

Both printf and scanf are just zipWith id.

```plaintext
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:

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

Hence, printf is a fold:

```plaintext
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 \texttt{printf} and \texttt{scanf} are just \texttt{zipWith id}.

\begin{verbatim}
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)
    = consS (d x) (printf ds xs)
\end{verbatim}

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

\begin{verbatim}
fold z g nil = z
fold z g (cons x xs) = g x (fold z g xs)
\end{verbatim}

Hence, \texttt{printf} is a fold, and the descriptor can be deforested:

\begin{verbatim}
printf = id
nilD nilA = nilS
consD d ds (consA x xs) = consS (d x) (ds xs)
\end{verbatim}
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 \texttt{printf} and \texttt{scanf} are just \texttt{zipWith \textit{id}}.

\begin{verbatim}
printf nilD nilA = nilS
printf (consD d ds) (consA x xs)  
    = consS (d x) (printf ds xs)
\end{verbatim}

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

\begin{verbatim}
fold z g nil = z
fold z g (cons x xs) = g x (fold z g xs)
\end{verbatim}

Hence, \texttt{printf} is a fold, and the descriptor can be deforested:

\begin{verbatim}
printf = id  scanf = id
nilD () = ()
consD d ds (x, xs) = (d x, ds xs)
\end{verbatim}

Choose tuple representation. Same with \texttt{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`. specification

implementation

```
scanf printf
```
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`

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

Fuse each primitive descriptor with `consD`.

```lit_str 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`

```plaintext
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.

```plaintext
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`

\[
\text{nilD} \quad () \quad = \text{nilA}
\]
\[
\text{consD} \quad d \quad = \quad d
\]

Fuse each primitive descriptor with `consD`.

\[
\text{lit str ds (s,ss)} \quad = \quad \text{consA} \quad (\text{assert (str = s)}) \quad (\text{ds ss})
\]
\[
\text{char ds (s,ss)} \quad = \quad \text{consA} \quad (\text{char_of_string s}) \quad (\text{ds ss})
\]
\[
\text{int ds (s,ss)} \quad = \quad \text{consA} \quad (\text{int_of_string s}) \quad (\text{ds ss})
\]

Primitive descriptors can consume and produce different amounts.

\[
\text{lit str ds inp}
\]
\[
= \quad \text{if String.length str} \quad \leq \quad \text{String.length inp} \quad \&\& \quad \\
\text{str} \quad = \quad \text{String.sub inp 0 (String.length str)} \quad \\
\text{then ds (String.sub inp (String.length str)} \quad \\
\text{(String.length inp - String.length str))} \quad \\
\text{else failwith "scanf lit"}
\]
On to the standard `scanf`

```plaintext
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`

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

Fuse each primitive descriptor with `consD`.

```ocaml
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.

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

Done!
Not quite the standard `printf`

We have:

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

We want:

```plaintext
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).

```
```
On to the standard printf

Begin by symmetry with scanf:

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

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

```plaintext
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`:

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

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

```plaintext
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`:

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

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

```haskell
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`.

```haskell
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`:

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

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

```plaintext
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.

```plaintext
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`:

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

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

```plaintext
lit str ds (xs) = str ^ ds xs
cchar 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!

```plaintext
lit str ds k
    = ds (fun s -> k (str ^ s))
cchar 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:

```ml
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:

```plaintext
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”:

```plaintext
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.