CS 320: Compilers

Pretty printing is pretty important:

- The opposite of parsing is *pretty printing*, which takes an abstract syntax tree and prints it out in a legible form.
  - Designing a good pretty printer is an important part of compiler design.
  - The pretty printer is used to report information about the location and nature of errors to the programmer.
  - The pretty printer helps the compiler writer debug their compiler.
- Unfortunately, we don't have time to cover techniques for developing general printing algorithms.

Pretty Printing

We do have time for something related but much simpler!

We will design a simple type-safe formatter resembling C's `sprintf` including

- Element Specifiers (`%d`, `%f`, etc.)
- Glue to concatenate specifications together
- Formatting Combinators

This lecture should also introduce you to the idea of developing a system of *combinators*, which are useful in designing domain-specific programming languages and libraries.

Composition of Functions

SML has a composition operator, $\circ$, defined as:

$$\boxed{\textbf{fun } (f \circ g)\ x = f(g(x))}$$

Clearly, if f has type $\beta \to \gamma$ and g has type $\alpha \to \beta$ then $f \circ g$ has type $\alpha \to \gamma$.

Give the type for $f \circ g$ if

(i) f : string $\to$ int, g : (int $\to$ bool) $\to$ string

(ii) f : (string $\to \alpha$) $\to$ (string $\to \alpha$),
g : (string $\to \alpha$) $\to$ string $\to$ int $\to \alpha$

Composition of Functions

SML has a composition operator, $\circ$, defined as:

$$\boxed{\textbf{fun } (f \circ g)\ x = f(g(x))}$$

Clearly, if f has type $\beta \to \gamma$ and g has type $\alpha \to \beta$ then $f \circ g$ has type $\alpha \to \gamma$.

Give the type for $f \circ g$ if

(i) f : string $\to$ int, g : (int $\to$ bool) $\to$ string

$f \circ g$: (int $\to$ bool) $\to$ int

(ii) f : (string $\to \alpha$) $\to$ (string $\to \alpha$),
g : (string $\to \alpha$) $\to$ string $\to$ int $\to \alpha$

Composition of Functions

SML has a composition operator, $\circ$, defined as:

$$\boxed{\textbf{fun}\ (f \circ g)\ x = f(g(x))}$$

Clearly, if f has type $\beta \to \gamma$ and g has type $\alpha \to \beta$ then $f \circ g$ has type $\alpha \to \gamma$.

Give the type for $f \circ g$ if

(i) f : string $\to$ int, g : (int $\to$ bool) $\to$ string

$f \circ g$: (int $\to$ bool) $\to$ int

(ii) f : (string $\to \alpha$) $\to$ (string $\to \alpha$),
g : (string $\to \alpha$) $\to$ string $\to$ int $\to \alpha$

$f \circ g$: (string $\to \alpha$) $\to$ string $\to$ int $\to \alpha$

Unparser
Description

We are going to develop an unparser that resembles the `printf` function in C. The function we develop, format, will:

- Interpret a *format* made up of *element specifiers*.
- Convert *arguments* to a *result* based on the format.
- Use the SML type-checker to catch bogus arguments (like inputting an integer where a string was expected).

Unparser
Description (Contd.)

- **Combinators**: We will implement only two combinators, -- for concatenation and $\gg$ for "piping" to a receiver function.

Note: our design will be flexible enough to allow us to implement combinators that perform operations like padding, fitting, etc. on the arguments easily, if we wanted to do so.

Unparser
Examples

Here are a few examples:

```
- format (L "PI ~ "--R) 3.1415
val it = "PI ~ 3.1415" : string
-
- val f = format (I--L " bottles of "--S--NL);
val f = fn : int -> string -> string
- f 99 "beer";
val it = "99 bottles of beer\n" : string
-
- format (I--NL>>TextIO.print) 10;
10
val it = () : unit
```

Unparser
Element Specifiers

$$\begin{aligned} \mathsf{I} &: (\text{string} \to \alpha) \to \text{string} \to \text{int} \to \alpha \\ \mathsf{B} &: (\text{string} \to \alpha) \to \text{string} \to \text{bool} \to \alpha \\ \mathsf{S} &: (\text{string} \to \alpha) \to \text{string} \to \text{string} \to \alpha \end{aligned}$$

- The first argument is a *continuation* that specifies what to do next.
- The second argument is the string result so far.
- The third argument is the argument that will actually get formatted.
- Simple usage: `I print ‘‘course: ’’ 320`

Unparser
Element Specifiers

The implementation is fairly easy. In the code below,

- k is the “continuation”: what to do after processing the current string.
- s is the string that has been produced so far.
- x is the argument that gets processed next.

$$\begin{array}{|l|} \hline \textbf{fun}\ \mathrm{I}\ k\ s\ x = k\ (s \wedge (\text{Int.toString}\ x)) \\ \\ \textbf{fun}\ \mathsf{B}\ k\ s\ x = k\ (s \wedge (\text{Bool.toString}\ x)) \\ \\ \textbf{fun}\ \mathsf{S}\ k\ s\ x = k\ (s \wedge x) \\ \hline \end{array}$$

Unparser
Glue

For “sticking” together element specifiers, we introduce glue specifiers. These are simpler than element specifiers:

$$\begin{array}{|l|} \hline \textbf{fun}\ \mathsf{NL}\ k\ s = k\ (s \wedge \text{“\textbackslash n”}) \\ \\ \textbf{fun}\ \mathsf{TB}\ k\ s = k\ (s \wedge \text{“\textbackslash t”}) \\ \\ \textbf{fun}\ \mathsf{L}\ x\ k\ s = k\ (s \wedge x) \\ \hline \end{array}$$

Unparser
Combinators

The simplest expression using piping is:

NL $\gg$ print : (unit $\to \alpha$) $\to$ string $\to \alpha$

where:

$$\begin{aligned} \mathsf{NL} &: (\text{string} \to \alpha) \to \text{string} \to \alpha \\ \text{print} &: \text{string} \to \text{unit} \end{aligned}$$

Assuming we want to do k next and we are given an input string s, this expression should concatenate a NL to s, print the resulting string and then do k. We want:

(NL $\gg$ print) k s = NL (k o print) s

In general:

$$\boxed{\textbf{fun}\ (f \gg g)\ k\ s = f\ (k \circ g)\ s}$$

Unparser
Combinators

The simplest expression using concatenation:

NL -- TB : (string $\to \alpha$) $\to$ string $\to \alpha$

where:

NL, TB : (string $\to \alpha$) $\to$ string $\to \alpha$

Assuming we want to do k next and we are given an input string s, then we should create the string $s \wedge$ NL $\wedge$ TB and then do k on the result. We want:

$$\begin{aligned}(\text{NL} \text{ -- } \text{TB})\ k\ s &= (\text{NL}\ (\text{TB}\ k))\ s \\ &= (\text{NL} \circ \text{TB})\ k\ s\end{aligned}$$

The concatenation combinator is composition

$\boxed{\textbf{val op -- = op} \circ}$

Unparser
Combinators (Contd.)

To see how the combinators and specifiers work, let us consider the format `I -- S -- NL >> print`.

We know that

$$\begin{aligned}\text{print} &: \text{string} \to \text{unit} \\ \text{NL} &: (\text{string} \to \alpha) \to \text{string} \to \alpha \\ \text{S} &: (\text{string} \to \alpha) \to \text{string} \to \text{string} \to \alpha \\ \text{I} &: (\text{string} \to \alpha) \to \text{string} \to \text{int} \to \alpha\end{aligned}$$

The type of `NL >> print` is

(unit $\to \alpha$) $\to$ string $\to \alpha$.

Unparser
Combinators (Contd.)

Proceeding similarly, we deduce the following types:

`S -- NL >> print` :
(unit $\to \alpha$) $\to$ string $\to$ string $\to \alpha$
`I -- S -- NL >> print` :
(unit $\to \alpha$) $\to$ string $\to$ int $\to$ string $\to \alpha$

Therefore, if `I -- S -- NL >> print` were to be given an initial continuation and an initial string, it would return a function of type int $\to$ string $\to \alpha$.

Unparser

The formatter "gets things going." All it needs to do is set up an initial continuation and an initial string:

$\boxed{\textbf{fun}\ \text{format}\ f = f\ (\textbf{fn}\ s \Rightarrow s)\ ""}$

Unparser

The Package So Far

```
structure Unparser = struct
    fun I k s x = k (s ∧ (Int.toString x))
    fun B k s x = k (s ∧ (Bool.toString x))
    fun R k s x = k (s ∧ (Real.toString x))
    fun S k s x = k (s ∧ x)
    fun C k s x = k (s ∧ (Char.toString x))
    fun NL k s = k (s ∧ "\n")
    fun TB k s = k (s ∧ "\t")
    fun L x k s = k (s ∧ x)

    infix 2 ≫
    infixr 1 --
    val op -- = op ∘
    fun (f ≫ g) k = f(k ∘ g)

    fun format f = f (fn s ⇒ s) " "
end
```


Unparsing

Combinators (Contd.)

Now we consider what exactly happens when we evaluate the expression

```
format (I -- S -- NL ≫ print)
       101 " dalmatians"
```

Using the definitions, we see that this is the same as

```
[I ∘ S ∘ (NL ≫ print)] (fn s ⇒ s) " "
       101 " dalmatians"
```

which can, in turn, be reduced to

```
(I ( S ( ((NL ≫ print) (fn s ⇒ s)))))
     " " 101 " dalmatians"
```

Lets call this result RR

Unparsing

Combinators (Contd.)

Of course, we know that

```
(NL ≫ print) (fn s ⇒ s) = NL print
```

Therefore, we see that RR is equivalent to

```
I (S (NL print)) " " 101 " dalmatians"
```

Now we use the definition of I to see that it "swallows" the empty string and 101, using the function (S (NL print)) as the continuation to give

```
S (NL print) "101" " dalmatians"
```


Unparsing

Combinators (Contd.)

This time we use the definition of S to see that it swallows the strings "101" and " dalmatian" using (NL print) as the continuation, yielding:

```
NL print "101 dalmatians"
```

Finally, we use the definition of NL to see that it merely calls the continuation (in this case, print) after tacking on a newline to the argument:

```
print "101 dalmatians\n"
```

which is exactly what we expected.