Standard ML on the Web server: Visualizing Lambda Calculus Reduction

Peter Sestoft (sestoft@dina.kvl.dk)
Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark

1996-11-18

1 Introduction

We show that World Wide Web (WWW) browsers provide an easy and portable way to create user interface for Standard ML (SML) programs, by writing the Standard ML applications as CGI[1] programs and running them on a WWW server. This also demonstrates that SML is a suitable language for CGI programming[2].

As an example interactive application we present a *lambda reducer*, a tool for visualizing reduction strategies for the pure untyped lambda calculus. Another example application (joint work with Ulla Dindorp) is a WWW-based browser for experimental data from agricultural research [3].

1.1 The lambda reducer

We use a WWW browser, such as Netscape Navigator, as the front-end of the lambda reducer, and run the reducer itself as a CGI program on a WWW server. The reducer is implemented in Standard ML, using the SML Basis Library [4] and the Moscow ML implementation [6]. The lambda reducer reads and parses a lambda term supplied by the user, and reduces it using the chosen reduction strategy and reduction tracer. It generates HTML[3] code which is sent by the WWW server to the browser on the user's machine. The browser interprets the HTML code and shows reduction traces etc. on the user's display.

The lambda reducer may be used as an educational tool, but we do not intend it to say anything new about the lambda calculus.

The lambda reducer is at `http://ellemose.dina.kvl.dk/~sestoft/lamreduce/`.

1.2 An example session

An example session with the lambda reducer is shown in Figure 1. In the upper frame, the user may enter a term and choose a reduction strategy and a reduction tracer, then click on the 'Do it' button. The response by the lambda reducer, usually a reduced term or a reduction trace, is then displayed in the lower frame. The S, K and I combinators, and a number of other abbreviations for lambda terms, are predefined.

In every term displayed, the current redex is highlighted and active: clicking on the redex will reduce that redex and display the resulting term, highlighting the next redex under the chosen strategy.

[1]CGI abbreviates Common Gateway Interface. A CGI program produces HTML code on request from a WWW browser and sends it to the browser.

[2]This observation is due to Jonas Barklund, Uppsala University.

[3]HTML stands for Hypertext Mark-up Language. HTML code instructs a WWW-browser to produce the lay-out, texts and images seen by the user.

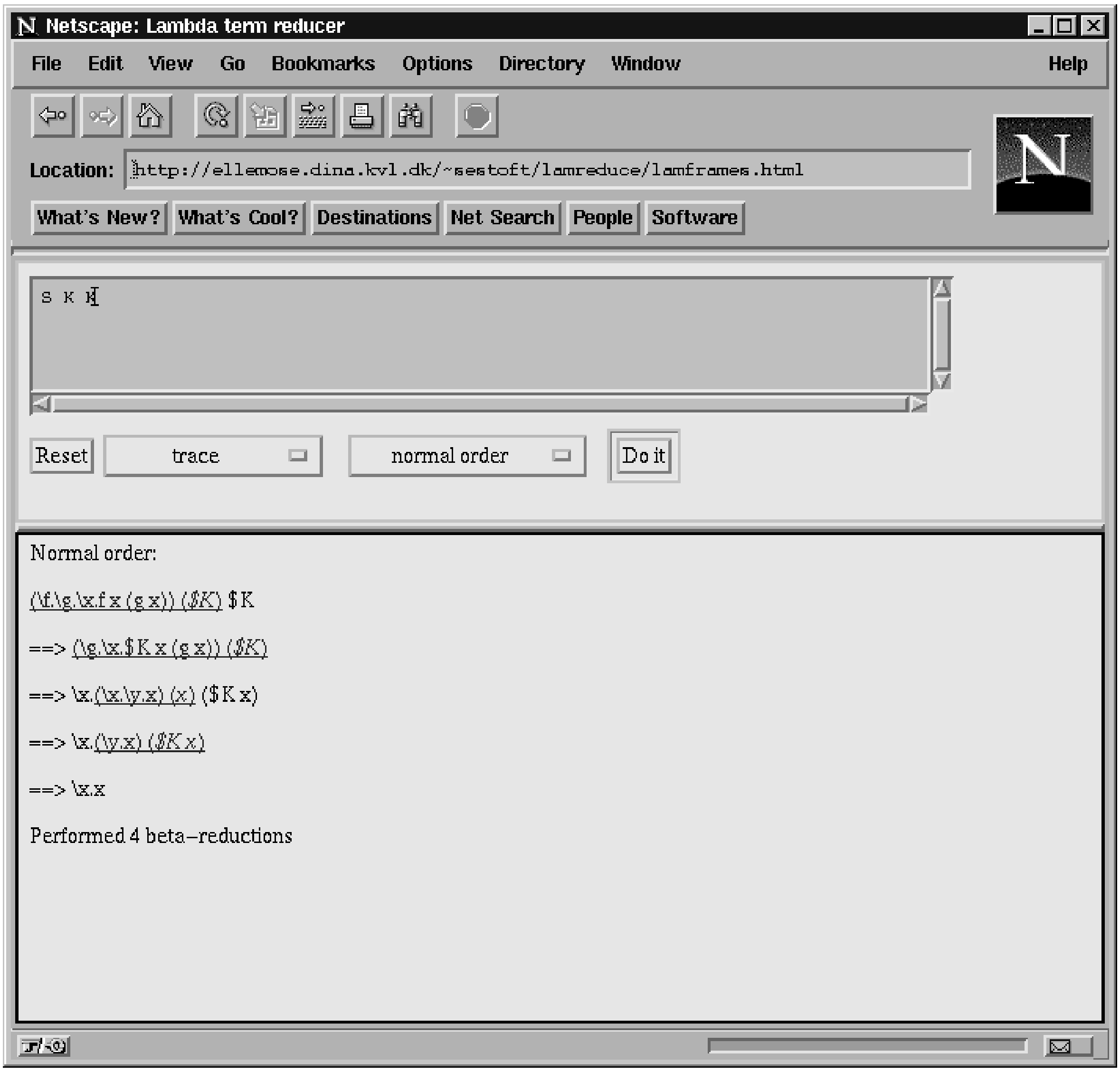

Figure 1: The lambda term reducer

2 Implementing reduction strategies for tracing

We consider the pure untyped lambda calculus, in which a term is a variable x, an abstraction $\lambda x.e$, or an application $(e_1\, e_2)$. The lambda terms may have free variables; these play the rule of constants or constructors in programming languages. We implement the following reduction strategies:

- call-by-name
- head spine reduction
- normal order
- call-by-value
- applicative order
- hybrid reduction

For brevity, we consider only the call-by-name strategy in this paper.

2.1 An example reduction strategy: call-by-name

The call-by-name strategy is leftmost outermost reduction to weak head normal form (whnf). We describe the reduction strategy by (big-step) operational semantics rules as shown below.

A variable x is in whnf and reduces to itself by rule Var. An abstraction $(\lambda x.e)$ is in whnf and reduces to itself by rule Lam.

An application $(e_1\, e_2)$ is reduced by first reducing e_1 to e_1'. If e_1' is an abstraction $(\lambda x.e)$, then $(e_1'\, e_2)$ is a redex, which is reduced by performing the substitution $e[e_2/x]$ (possibly renaming bound variables to avoid capture), and then reducing the resulting term (rule Appr).

If e_1' is not an abstraction, then $(e_1'\, e_2)$ is not a redex; consequently e_2 must be reduced to e_2' and the application $(e_1'\, e_2')$ must be returned (rule Appn). Namely, e_1' must be a head normal form (hnf), that is, a variable applied to zero or more terms, so $(e_1'\, e_2)$ is a hnf and a weak head normal form (whnf) too.

$$x \xrightarrow{bn} x \qquad \text{(cbn Var)}$$

$$(\lambda x.e) \xrightarrow{bn} (\lambda x.e) \qquad \text{(cbn Lam)}$$

$$\frac{e_1 \xrightarrow{bn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{bn} e'}{(e_1\ e_2) \xrightarrow{bn} e'} \qquad \text{(cbn Appr)}$$

$$\frac{e_1 \xrightarrow{bn} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{bn} (e_1'\ e_2)} \qquad \text{(cbn Appn)}$$

Rule Appn with the negative premise $e_1 \xrightarrow{bn} e_1' \not\equiv \lambda x.e$ can be thought of as an abbreviation for two positive rules, covering the cases $e_1' \equiv x$ and $e_1' \equiv (e_{11}\, e_{12})$.

2.2 A reducer for the call-by-name strategy

Representing a variable x by the data structure `Var x` (of type `lam`), an abstraction $\lambda x.e$ by `Lam(x,e)`, and an application $(e_1\, e_2)$ by `App(e_1, e_2)`, the operational semantics rules above can be implemented in Standard ML as follows:

```
fun cbn (Var x)        = Var x
  | cbn (Lam(x, e))    = Lam(x, e)
  | cbn (App(e1, e2)) =
    (case cbn e1 of
         Lam (x, e) => cbn (subst(e, e2, x))
       | e1'        => App(e1', e2))
```

The type of `cbn` is `lam -> lam`, and the intention is that `cbn` e `=` e' iff $e \xrightarrow{bn} e'$. It is clear that the `Var` and `Lam` branches of the function implement the semantics rules Var and Lam.

The rules Appr and Appn are implemented by the two cases of the `App` branch. In the first case, expression e_1 reduces to an abstraction $(\lambda x.e)$, the substitution $e[e_2/x]$ (with possible renaming) is performed by the call `subst(e, e2, x)`, and reduction by `cbn` continues from this expression, as prescribed by rule Appr. In the second case, e_1 reduces to a non-abstraction e_1', and the term $(e_1'\, e_2)$ is returned.

2.3 Traces, contexts and labels

The `cbn` reducer above performs the beta-reductions in the expected leftmost outermost order (because SML is call-by-value), and returns the final reduced term (or loops). However, the `cbn` function always focuses on some subterm e_{sub} of the entire current term e_{top}, but to trace the reduction, we want to display the entire current term e_{top}. For instance, when performing the beta-reduction

$$(x\,((\lambda y.y)\,z)) \xrightarrow{bn} (x\,z)$$

the reducer will focus on the redex $((\lambda y.y)\,z)$, but we want the tracer to display the reduction on the entire term, as above.

For this purpose, we define a *tracing reducer* `cbnc` by modifying `cbn`. The tracing reducer takes as argument a context $c[\,]$, which can reconstruct e_{top} from e_{sub}, and applies this context to the redex before every beta-reduction. This application will mark the redex and reconstruct the entire current term; if desired it may print the reconstructed term as a side effect, highlighting the redex. The context $c[\,]$ does not affect the reduction strategy in any way.

For marking the redex e, we use labelled terms e^i, where i is an integer[4].

As the reducer descends into the term, the context $c[\,]$ must be extended. If `c` in the program represents context $c[\,]$, then

`c o Lamx x`	is the context	$c[\lambda x.[\,]]$
`c o Lbli i`	is the context	$c[[\,]^i]$
`c o App2 e1`	is the context	$c[(e_1\,[\,])]$
`c o App1 e2`	is the context	$c[([\,]\,e_2)]$

[4]Lévy labels [1, appendix] could be used instead.

2.4 The tracing call-by-name reducer

The tracing reducer `cbnc`, of type `(lam -> unit) -> (lam -> lam)`, is

```
fun cbnc c (Var x)       = Var x
  | cbnc c (Lbl(i, e))   = Lbl(i, cbnc (c o Lbli i) e)
  | cbnc c (Lam(x, e))   = Lam(x, e)
  | cbnc c (App(e1, e2)) =
    findlam (cbnc (c o App1 e2) e1)
            (cbnc c o csubst c e2)
            (App1 e2)
```

A variable x reduces to itself. A labelled term e^i is reduced by reducing e and labelling the result with i. The reduction of e takes place in the context $c[[\,]^i]$, that is, the current context c extended with the labelling i. An abstraction $\lambda x.e$ reduces to itself.

An application $(e_1\, e_2)$ is reduced by the call to auxiliary function `findlam`; this function implements the choice between the Appr and Appn rules in the operational semantics of Section 2.1.

First, the argument `(cbnc (c o App1 e2) e1)` is evaluated to reduce e_1 in the context $c[([\,]\, e_2)]$, possibly producing some reduced term e_1'. (The second and third arguments are partial applications and their evaluation does not involve any calls to `cbnc`).

If e_1' is a (possibly labelled) abstraction $(\ldots(((\lambda x.e)^{i_1})^{i_2})\ldots)^{i_n}$, $n \geq 0$, then `findlam` strips the marks off the abstraction and applies `(cbnc c o csubst c e2)` to the pair $(e_1', \lambda x.e)$. This causes function `csubst` to build the labelled redex $(e_1'\, e_x)^i$ and apply the context function c to it for tracing. Then `csubst` performs the substitution $e[e_2/x]$ (with possible renaming), and the result is passed to `cbnc` for further reduction in the original context $c[\,]$.

If e_1' is not an abstraction, then `findlam` applies `(App1 e2)` to e_1', producing the term $(e_1'\, e_2)$.

Note that the context and the subterm 'add up' to the entire current term in all recursive calls to `cbnc`.

3 Implementing the reduction tracers

There are several functions for tracing the reduction of a lambda term:

- trace the reduction sequence, highlighting current redex
- normalize the term, and count the number of beta reductions performed
- trace the reduction sequence, labelling all redexes and the contracted terms
- singlestep through the reduction sequence

These tracers can be used with all the reduction strategies previously mentioned. A tracer must be called with a limit `max` on the number of beta-reductions to perform; this is necessary for limiting the server resources consumed by remote users.

Here we describe only the full-trace and the normalizing tracer; Section 4.6 describes the single-stepping tracer.

3.1 Tracing: print all steps of the reduction

The tracer `trace` runs a reducer, printing the current term before every beta-reduction:

```
fun trace max reduce e =
    let val incr = stopat max
        fun ppreduces e = (Format.ppp e; print "<br>==> "; incr e)
        val e' = reduce ppreduces e
    in Format.ppp e'; showsteps () end
    handle Enough _ => exceeded max
```

Function `stopat` creates a function `incr` for counting the number of reduction steps. This function will raise exception `Enough` with the current term when `max` beta-reductions have been performed.

Function `ppreduces` is the initial context function; when applied to a lambda term, it invokes function `Format.ppp` to pretty-print it (see Section 4.4), issues an HTML linebreak `<br>`, and increments the step counter.

The function `reduce`, which might be the call-by-name reducer `cbnc` from Section 2.4, is applied to the initial context function `ppreduces` and the term e to be reduced. As described in Section 2.3, the `ppreduces` function will be invoked by `reduce` before every beta-reduction, thus printing the current term and counting the number of beta-reductions. If the term reaches a normal form e' (by `reduce`) in at most `max` steps, then e' and the total step count are printed (by function `showsteps`). If more than `max` steps are required, then reduction stops because `incr` raises `Enough`, and an error message is printed.

For example, with $\Omega = ((\lambda x.(x\,x))\,(\lambda x.(x\,x)))$, the application `trace 100 cbnc` Ω will print 100 steps of the infinite reduction sequence $\Omega \xrightarrow{bn} \Omega \xrightarrow{bn} \Omega \xrightarrow{bn} \ldots$, then terminate with an error message.

3.2 Displaying the reduced term

The tracer `show` runs the reducer, counting the number of beta-reductions, and prints the final term together with this count:

```
fun show max reduce e =
    let val incr = stopat max
        val e' = reduce incr e
    in Format.ppp e'; showsteps () end
    handle Enough e'' => (exceeded max; showlast e'')
```

The main difference from `trace` is that the initial context function is `incr`, which simply discards the intermediate terms.

If the term reaches a normal form e' (by `reduce`) in at most `max` steps, then e' and the total step count are printed (by function `showsteps`). If more than `max` steps are required, then the last term e'' produced will be printed, by handling the exception `Enough` raised by `incr`.

4 Using a WWW browser as front-end for interactive programs

4.1 Basic machinery

The *client* is the user's computer, running the WWW browser (e.g., Netscape Navigator); the *server* is the remote computer, running the WWW services, including the lambda reducer.

The cycle of interaction between the user and the CGI program (the lambda reducer) is the following:

1. The user clicks on a hyperlink or ‘Do it’ button inside WWW browser on client.
2. An HTTP request is sent to the WWW server.
3. The server invokes the requested CGI program with the given parameters.
4. The CGI program parses the parameters; performs some computation; generates HTML code.
5. The server sends the HTML code to client.
6. The WWW browser on the client interprets the HTML code received and displays text.

If the CGI program in step (4) is the lambda reducer, then the following actions are performed as part of step (4):

4.1 Get the CGI parameters, specifying the lambda term, reduction strategy, and reduction tracer.
4.2 Scan and parse the lambda term.
4.3 Perform and trace the reduction of the lambda term, possibly printing intermediate terms in HTML-format to standard output.

Note that the CGI program is invoked to compute the response to a single request; hence no state is preserved from one invocation to the next one. However, some state information may be encoded in the (hyperlinks of the) response sent from the server to the client, to be sent back to the server along with the next request from the client. An example of this is given in Section 4.6.

The data entry area in the upper half of Figure 1 is created by a ‘form’ element, a piece of HTML code. The form specifies the name of the server to which to send the completed form (when the user clicks on ‘Do it’), and which CGI program on the server to invoke to process the form.

4.2 Wordseq: Efficiently concatenable strings

We want the ability to generate HTML code compositionally. For instance, we may want to generate the contents s of an HTML page (or table, etc.), and subsequently wrap the required mark-up around it, as in "<BODY>" ^ s ^ "</BODY>", to create an HTML page.

However, wrapping such ‘parentheses’ around a given string s in this naive way requires copying the entire string twice. In addition to the work involved in copying, this requires allocation of ever larger strings, may cause fragmentation in the ML heap, and hence may put much load on the garbage collector.

A simple solution is to introduce a type of word sequences which permits constant-time concatenation at the expense of some space overhead. A word sequence is empty, or consists of a single string, or is the concatenation of two word sequences:

```
datatype wordseq =
    Empty
  | $ of string
  | && of wordseq * wordseq
```

A word sequence may be printed to standard output (as a side effect) by a simple tree traversal. Now we can have a non-copying constant-time operation `val prpar: wordseq -> wordseq` to surround a word sequence by parentheses:

```
fun prpar s = $ "(" && s && $ ")"
```

There are other possible solutions to this problem, which may have less space overhead: (1) a language in which string concatenation is lazy by default (implicitly delayed evaluation), and (2) a functional representation of the strings (explicitly delayed evaluation).

Alternative (1) reduces space consumption only if the language as a whole uses lazy evaluation, in which case extra care must be taken to avoid laziness where it is not needed. Alternative (2) does not allow the examination of a string already built, an operation which is useful *e.g.* for avoiding excess parentheses around a term.

The word sequence solution works well in practice, probably because the amount of text displayed in a WWW browser window usefully is less than 100 KB.

4.3 HTML as the application program's interface

Using word sequences, various HTML mark-up functions are easily written in SML:

```
val mark0    : string -> wordseq -> wordseq
val mark1    : string -> wordseq -> wordseq -> wordseq
val href     : wordseq -> wordseq -> wordseq
val htmldoc  : wordseq -> wordseq -> wordseq
val cgiencode: string -> wordseq
val cgicall  : string -> (string * wordseq) list -> wordseq
...
```

4.4 Printing lambda terms with hyperlinks from redexes

When the `Format.ppp` function prints a lambda terms, it creates special mark-up for the redex, so that clicking on the redex will reduce it and print the resulting term. This is useful for singlestepping the reduction of the term; see Section 4.6.

The markup for invoking a CGI program on the WWW server resembles that for loading an HTML file, except that the file name is replaced by the program name, followed by a question mark '?', followed by program arguments:

```
<a href="http:lamreduce?action=singlestep&expression=S+K+K
                        &evalorder=normal+order&stepno=1">
```

(Linebreak inserted for readability). This markup invokes the CGI program `lamreduce`, that is, the lambda reducer. It passes four arguments, bound to the parameters `action`, `expression`, `evalorder`, and `stepno`. The argument values are encoded (where necessary) by replacing space with `+`; thus `S+K+K` means `S K K`.

This markup is generated by the following SML program fragment:

```
Html.cgicall "lamreduce"
     [("action",     action),
      ("expression", exprws),
      ("evalorder",  evalordws),
      ("stepno",     $ (Int.toString (1 + Lambda.stepCount())))]
```

4.5 Accessing the parameters of CGI programs

The CGI program obtains the values of its parameters, such as `expression`, from the environment. The `mosmlcgi` library [2] created by Jonas Barklund, Uppsala, provides access to the parameters (using `OS.Process.getEnv` from the Basis Library). For instance,

```
mosmlcgi.cgi_field_string "expression"
```

returns `SOME` e if the CGI program was invoked with an argument specification `expression=`e.

4.6 Singlestepping reduction

The single-stepping tracer (cf. Section 3) called `single` performs at most `stepno` beta-reductions, and displays the resulting term. To implement this we create a function `incr`, which will raise `Enough` with the current expression after `stepno` beta-reductions. We reduce the term e using that initial context function, and arrange to handle `Enough` and print the term e' passed with it, when `stepno` beta-reductions have been performed:

```
fun single stepno reduce e =
    let val incr = stopat stepno
        val e'   = reduce incr e
    in Format.ppp e' end
    handle Enough e' => Format.ppp e';
```

In the term produced after n beta-reductions, the redex (if any) is marked up with a call to the singlestepping tracer, whose step count is set to $n+1$. Hence a click on the redex (in the browser window) will perform one more beta-reduction and display the result. The perceived effect is that repeated clicking on the redex singlesteps through the reduction sequence, although in fact the reduction start from scratch, and just proceeding one step further than before.

For example, this is the sequence of marked-up terms produced by single-stepping normal order reduction of the term $S\,K\,K$, where $S \equiv \lambda f.\lambda g.\lambda x.f\,x\,(g\,x)$ and $K \equiv \lambda x.\lambda y.x$. Note that the `expression` passed to `lamreduce` is `S K K` in all five steps; only the `stepno` changes from invocation to invocation:

```
<a href="lamreduce?
action=singlestep&expression=S+K+K&evalorder=normal+order&stepno=1">
(\f.\g.\x.f x (g x)) (K)</a> K

<a href="lamreduce?
action=singlestep&expression=S+K+K&evalorder=normal+order&stepno=2">
(\g.\x.K x (g x)) (K)</a>

\x.<a href="lamreduce?
action=singlestep&expression=S+K+K&evalorder=normal+order&stepno=3">
(\x.\y.x) (x)</a> (K x)

\x.<a href="lamreduce?
action=singlestep&expression=S+K+K&evalorder=normal+order&stepno=4">
(\y.x) (K x)</a>

\x.x
```

5 Evaluation

5.1 Size of the implementation

The complete lambda reducer, implementing six reduction strategies and four tracers, consists of five SML modules, plus specifications of the lexical and syntactic structure of the input language (which may contain comments and global abbreviations, not discussed in this paper):

Module	Signature	Implementation	Contents
`Env`	13 lines	59 lines	Global abbreviations
`Format`	8 lines	81 lines	Making hyperlinks from redexes
`Lambda`		120 lines	Terms, substitution, abbreviations
`Main`		65 lines	Argument parsing, invoke reducers
`Reducers`	24 lines	193 lines	Reduction strategies and tracers
Lexer specification		53 lines	Lexical (comments, identifiers, ...)
Parser specification		55 lines	Syntax (user-defined abbreviations)

The total size of the source is 671 lines or 21 KB. In addition, we use the general modules `Wordseq` (32 line signature and 84 line implementation) and `Html` (31 line signature and 69 line implementation).

5.2 Speed

CGI programs written in Moscow ML load fast and give good interactive response. There are two reasons for this: the runtime system is small, and the programs are precompiled to compact bytecode files.

The size of the Moscow ML runtime system (which is closely based on the Caml Light runtime system [5]) is 75 KB, as compared to the Perl interpreter's 520 KB (for the Solaris 2 operating system).

Moscow ML programs are compiled to compact bytecode files; the size of the bytecode file for the lambda reducer is 27 KB. A Perl script, on the other hand, is stored in source form and is parsed and compiled to an internal form on every invocation, causing a certain delay at every invocation.

5.3 General-purpose programming language versus scripting languages

The traditional scripting languages provide powerful functions for string manipulation, but these are typically line-oriented and do not handle recursive structures well. We have found that neat Perl scripts, which 'almost' solve the problem at hand, frequently grow into a complicated tangles of fixes, and finally have to be discarded.

Standard ML is a general purpose programming language, and provides the means for combining solutions to the various tasks facing the CGI programmer: higher order functions, user-defined data structures, parametric polymorphism, the SML Basis Library with support for string manipulation, a module system permitting separate compilation and top-down as well as bottom-up program development.

String manipulation, associative arrays, access to CGI parameters, automatic storage management, and fast turn-around (because no need to compile and link) are some of the reasons Perl scripts are prevalent for CGI programming. We have found that using a light-weight SML implementation supported by suitable libraries is no harder and leads to robust, reliable, and maintainable CGI programs.

5.4 Using WWW and HTML for creating visual user interfaces

WWW browsers and the HTML language offer a simple way to create visual user interfaces for Standard ML programs. No special operating system support is needed, only the ability to print to standard output, and the ability to access environment variables (needed by the `mosmlcgi` package). Both are provided by the language and the Basis Library.

HTML is platform independent, and freely licensed WWW browsers exist for almost all platforms (Unixes, Windows'95, MacOS, OS/2).

Some platform-independent packages for developing visual user interfaces exist, but they are usually tied to particular languages, such as C, C++, or Common Lisp. To our knowledge, there are no such packages for Standard ML at present.

The present simple approach to creating visual user interfaces is portable, and perhaps even durable: although currently evolving, HTML is being standardized, and we will probably use HTML five or ten years from now. By contrast, many platform- and language-specific window application program interfaces are likely to be obsolete by that time.

Our approach has some limitations, too:

- If the user needs to modify data stored on the WWW server, a number of security and resource issues must be addressed, which are rather similar to those of mainframe installations. A closer integration with the WWW browser, perhaps by exploiting applets, is needed for the user to store and modify data locally.
- Considerable network and server resources are required, compared to the visual user interfaces running on the user's personal computer.
- Using HTML provides less control over the actual appearance of windows etc. on the user's display than do traditional tools for creating visual user interfaces; this is the down-side of platform independence.

6 Related work

One source of inspiration is Olin Shivers's Scheme shell `scsh`, in which Shivers intended Scheme to replace the 'little languages' such as awk, perl, sed, and shell scripting languages [8]. The little languages have a number of syntactic quirks, and usually lack suitable means of program structuring and integration.

Another source of inspiration is Jonas Barklund's summer course *The Internet and Programming* given at Uppsala University, in which he used SML and the Moscow ML implementation. In that connection he created the `mosmlcgi` library which provides access to the arguments of CGI scripts [2].

7 Conclusion

We have found that Standard ML programs can be given visual user interfaces by exploiting current WWW browser technology and the HTML language. With a modest programming effort this can improve the appeal and usability of some Standard ML programs considerably.

One possible application area is educational software, where the ability to make updates centrally, and to create a single version for all platforms, is useful.

Future work The lambda reducer may be adapted to visualize reduction in a lazy (call-by-need) version of the lambda calculus, most likely based on a suitable abstract machine [7].

The `Html` and `Wordseq` libraries should be overhauled and packaged up for distribution. For other applications we plan add forms-based file upload to the `mosmlcgi` library.

Acknowledgements We are indebted to Jonas Barklund for the inspiration provided by his use of SML in the summer course *The Internet and Programming*, and also for his `mosmlcgi` library.

References

[1] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[2] J. Barklund. The mosmlcgi library. Web page at `http://www.csd.uu.se/~jonas/mosmlcgi/`.

[3] U. Dindorp and P. Sestoft. The sfd dataset browser. overview and technical description. Technical report, Royal Veterinary and Agricultural University, Denmark, 1996. Draft 0.07.

[4] E. Gansner and J. Reppy. Standard ML Basis Library. Technical report, AT&T Research, 1996.

[5] X. Leroy. The Zinc experiment: An economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, France, 1990.

[6] S. Romanenko and P. Sestoft. *Moscow ML Owner's Manual, version 1.41*, October 1996. Available at `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[7] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3), May 1997. (To appear). Available at ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Sestoft/papers/amlazy5.ps.gz.

[8] O. Shivers. A scheme shell. Technical Report TR-635, Laboratory for Computer Science, MIT, 1994. To appear in Journal of Lisp and Symbolic Computation.