

WebScript, a stack-oriented programming language

- Programming in WebScript
 - WebScript is a descendant of Postscript, a stack-oriented dynamically typed language inspired by Forth.
 - A type system for WebScript?
- This is a crude draft.

Postscript and WebScript programs

Programs are written in postfix (or reverse Polish) notation, using only the evaluation stack and dictionary stack. All operators take their arguments from the stacks and leave their results on the stacks. A new operator is defined by binding it in a dictionary.

Examples:

- Add 67 and 23 and leave the result on the stack:
67 23 add
- Bind variable port (in the current dictionary) to the string (8080):
/port (8080) def
- Access variable port (in the current dictionary):
port
- Define addfour (in the current dictionary) as the function which adds 4 to its argument:
/addfour { 4 add } def
- Apply addfour to 117 twice (giving 125):
117 addfour addfour

Datatypes in WebScript and Postscript

- Booleans: false, true
- Numbers: 45
- Literal names: /foo
- Text strings: (foo), (This is a string\n)
- Arrays: [23 false [56 (foo)]]
- Dictionaries: << /port (8080) /host (130.225.40.253) >>
This is the dictionary that maps literal name /port to the string (8080) and literal name /host to the string (130.225.40.253).
- Procedures: { 4 add }
This is the procedure which adds 4 to its argument.

Data structures accessible to an executing program:

- Operand stack: a stack of values
 - Dictionary stack: a stack of dictionaries
- The *current dictionary* is the one on top of the dictionary stack.

Example function definitions

- Define function printn which prints 1, 2, ..., n (Postscript's print function is =):
/printn { 1 1 3 -1 roll { = } for } def
- Define function sumn which computes 1 + 2 + ... + n:
/sumn { 0 1 1 4 -1 roll { add } for } def
- Define function rown which prints row i of an n x n HTML multiplication table:
/rown { exch /n exch def (<TR>) = 0 1 3 -1 roll { (<TD>) = n mul = } for }
- Define function tablen which prints an n x n HTML multiplication table:
/tablen { /n exch def (<TABLE>) = 0 1 nn { nn rown } for (</TABLE>) = } d

Practical experiments with Webscript and Postscript

- Use the webnet server (<http://webnet.it-c.dk>, accessible from the student net). Follow the link called WebScript Debugging Tool, and log in using the `admin` account.
 - Use the free Postscript interpreter Ghostscript (on any computer). Of course, this does not allow one to experiment with Webscript-specific extensions.
 - Using a Unix/Linux terminal window: type `gs -dNODISPLAY`
 - Using the ITC MS Windows student machines, and a Command Prompt (DOS box), type
`C:\Aladdin\gs6.01\bin\gswin32 -dNODISPLAY`
or `C:\Aladdin\gs6.01\bin\gswin32c -dNODISPLAY`
- For a more convenient setup, run Ghostscript inside a shell inside GNU Emacs (under Unix or MS Windows). Note that Webscript's polymorphic `xprint` function is called `==` in Postscript.

```
sestoff@jones > gs -dNODISPLAY
GNU Ghostscript 5.10 (1998-12-17)
Copyright (C) 1997 Aladdin Enterprises, Menlo Park, CA. All rights reserved.
This software comes with NO WARRANTY: see the file COPYING for details.
GS>56 7 add ==
63
GS>/f { 2 add } def
GS>67 f ==
69
GS>67 f f ==
71
GS>/fac { dup 0 eq { pop 1 } { dup 1 sub fac mul } ifelse } def
GS>3 fac ==
6
GS>7 fac ==
5040
GS>10 fac ==
3628800
GS>0 1 7 { fac == } for
1
1
2
6
24
120
720
5040
GS>quit
sestoff@jones >
```

Static type checking for Webscript – why?

Forth, Postscript and Webscript are dynamically typed. Every operator (such as `add`) checks at run-time that it is applied to arguments of the right type. An error is signalled when this is not the case.

Advantages of static type checking

Static type check (at compile time) provides a first check that the program makes any sense at all. Catches errors early: during development rather than when deploying the program. Saves time, money and embarrassment. Statically inferred types provide a scaffold for other information, such as bounded stack depth, or sized types.

Strategic value to the RCES project

This is the kind of work I'd like to do. Forth is being used in embedded systems (e.g., in space). Our ideas might (just might) improve Forth practice.

Related work: Typed Forth

Bill Stoddart and Peter J Knaggs: Type Inference in Stack Based Languages, Formal Aspects of Computing (1992) 3. Presents a type algebra in which multiplication is function composition, and addition is alternative types (overloading). The zero is the inconsistent type ϕ , the unit is the type $(\rightarrow)$, the type which takes nothing from the stack and leaves nothing. Not implemented.

Stephan Becher: <http://home.t-online.de/home/s.becher/forth>, September 2000. Provides overloading and subtyping, an implementation available (for MS Windows), no theory.

Related work: Typed assembly language with stacks

Greg Morrisett, Karl Cray, Neal Glew, David Walker: Stack-Based Typed Assembly Language. Types in Compilation, Kyoto, Japan, March 25-27, 1998. Springer-Verlag LNCS 1473. At the research frontier of mainstream type inference. Requires polymorphic recursion (undecidable) in the stack type; handles stack concatenation through the use of an additional pointer into the stack, requires some linear logic hack. Implementation available.

Related work: Typed stack-based languages

Robin Popplestone: Specifying Types by Grammars: A Polymorphic Static Type Checker Operating at a Stack Machine Level. Glasgow FP Report, 1994. Intended application areas: POR-11, Postscript, Forth. The description is biased towards the implementation in Prolog (a rather poor one, it seems), not mainstream. [A companion paper presents more theory; I haven't read it yet.]

Related work: Dependent types

Frank Pfenning and Weihong Xi: Eliminating array bound checking through dependent types. ACM SIGPLAN '98 conference on Programming Language Design and Implementation, June 1998, Montreal Canada, 249-257.
There's much more.

Related work: Sized types

Hughes, J., Pareto, L., and Sabry, A. Proving the correctness of reactive systems using sized types. In ACM Symposium on Principles of Programming Languages (1996).

W.-N. Chin and S.-C. Kuo. Calculating sized types. ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 62–72. ACM, New York, Jan. 2000

Sketch of type rules for WebScript/Postscript

The stack top is to the right, following WebScript and Forth conventions.

t	a type
lit	the type of statically known literals (e.g. $/\text{Foo}$)
int	the type of integers
int_n	the singleton type containing the integer n
σ	a stack (sequence) of value types
δ	a stack (sequence) of dictionary types
$t[n]$	the type of uniform arrays of length n and with element type t
dty	a dictionary type, similar to SML record type
ply	a procedure type, similar to SML function type

A dictionary type dty is a finite map from literals to types.

A procedure type ply is a universally quantified pair of an argument type and a result type.

The empty sequence is written $[]$. The post-cons operator $s :: x$ extends the sequence s with element x .
The associative append operator creates the concatenation $s1 @ s2$ of two sequences.

Typing judgments

There are no named entities in the language except those bound in the dictionaries.
Hence the typing rules have no context.

The general form of a judgement is :

$$\vdash iseq : \sigma_1, \delta_1 \longrightarrow \sigma_2, \delta_2$$

where $iseq$ is an instruction sequence, the σ_i are operand stacks, and the δ_i are dictionary stacks.

Dictionary stack convention (for brevity): a judgement

$$\vdash t : \sigma_1 \longrightarrow \sigma_2$$

which does not mention the dictionary stack is understood to abbreviate the judgement

$$\vdash t : \forall \delta. \sigma_1, \delta \longrightarrow \sigma_2, \delta$$

which does not modify the dictionary stack δ .

This is similar to the state passing convention in the SML operational semantics.

$$\begin{aligned} \vdash \text{abs} : \sigma :: int \longrightarrow \sigma :: int \\ \vdash \text{add} : \sigma :: int :: int \longrightarrow \sigma :: int \\ \vdash \text{aload} : \sigma :: t[n] \longrightarrow \sigma :: t :: \dots :: t :: t[n] \end{aligned}$$

where the number of ts on the right-hand side is n .

$$\begin{aligned} \vdash \text{and} : \sigma :: bool :: bool \longrightarrow \sigma :: bool \\ \vdash \text{and} : \sigma :: int :: int \longrightarrow \sigma :: int \\ \vdash \text{array} : \sigma :: int_n \longrightarrow \sigma :: t[n] \\ \vdash \text{astore} : \sigma :: t :: \dots :: t :: t[n] \longrightarrow \sigma :: t[n] \end{aligned}$$

where the number of ts on the left-hand side is n .

$$\begin{aligned} \vdash \text{begin} : \sigma :: dty, delta \longrightarrow \sigma, delta :: dty \\ \vdash \text{cleartomark} : \sigma_1 @ mark :: \sigma_2 \longrightarrow \sigma_1 \\ \vdash \text{clear} : \sigma \longrightarrow [] \end{aligned}$$

$\vdash \text{end} : \sigma, \delta :: dty \longrightarrow \sigma, \delta$
 $\vdash \text{forall} : \sigma :: t[n] :: pty \longrightarrow \sigma$
 where $pty = \sigma :: t \longrightarrow \sigma$ (this is SML app)
 $\vdash \text{forall} : \sigma :: t1 :: t2[n] :: pty \longrightarrow \sigma :: t1$
 where $pty = \sigma :: t1 :: t2 \longrightarrow \sigma :: t1$ (this is SML fold!, more general types are possible)
 $\vdash \text{for} : \sigma :: int :: int :: int :: pty \longrightarrow \sigma$
 where $pty = \sigma :: int \longrightarrow \sigma$ (plain for-loop)
 $\vdash \text{for} : \sigma :: t :: int :: int :: int :: pty \longrightarrow \sigma :: t$
 where $pty = \sigma :: t :: int \longrightarrow \sigma :: t$ (accumulating for-loop)
 $\vdash \text{get} : \sigma :: dty :: lit \longrightarrow \sigma :: t$
 when $(lit, t) \in dty$
 $\vdash \text{if} : \sigma_1 :: bool :: pty \longrightarrow \sigma_2$
 when $pty = \sigma_1 \longrightarrow \sigma_2$
 $\vdash \text{ifelse} : \sigma_1 :: bool :: pty :: pty \longrightarrow \sigma_2$
 when $pty = \sigma_1 \longrightarrow \sigma_2$

Challenges and open problems (lots)

- Need dependent types int_n = the type containing only the integer n .
- Need arrays of a specific size: $\{n\}$ is an array of length n
- Need stacks of a specific size, or consistent pointers into the stack (Morrisett et al).
- Need polymorphic recursion to typecheck recursively defined functions.
- Some functions can be usefully typed only in context, using a type rule. For instance:
 $\vdash \text{search } \text{if} : \sigma_1 :: pty :: string :: string \longrightarrow \sigma_1 @ \sigma_2$
 where $pty = \sigma_1 :: string :: string \longrightarrow \sigma_1 @ \sigma_2$
 and $ptyf = \sigma_1 :: string \longrightarrow \sigma_1 @ \sigma_2$
- Postscript arrays may be inhomogeneous, in a type system one must disallow this or require constant indexes. Perhaps distinguish homogeneous arrays from inhomogeneous tuples or records, as in Standard ML:
 - homogeneous arrays when element type t and indexing by arbitrary index expressions;
 - inhomogeneous arrays with different element types, and indexing by constant indexes only.