An O'small Interpreter Based on Denotational Semantics

Andreas V. Hense

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 07/91

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication and will probably be copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the publisher, its distribution prior to publication should be limited to peer communication and specific requests.

An O'SMALL Interpreter Based on Denotational Semantics

Andreas V. Hense

Universität des Saarlandes
Im Stadtwald 15
6600 Saarbrücken 11, Germany
hense@cs.uni-sb.de

October 31, 1991

Abstract

An interpreter for the object-oriented programming language O'SMALL is presented. It consists of a translation of a denotational semantics into the functional programming language Miranda. A parser and the ftp-location of the relevant files are also provided.

keywords: denotational semantics, interpreter, object-oriented programming

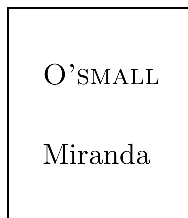
1 Introduction

When we call a problem trivial or a standard exercise it usually means that we don't want to solve it because there are – at least there should be – people for who it is easy to do. This report contains the solution of the standard exercise:

Given the denotational semantics of a language write an interpreter for it!

You may say "that's not trivial at all, you need these clever techniques for writing efficient interpreters, you have to think of the appropriate data structures,... ". But who said it has to be efficient? All we want is a prototype and we will see that with the choice of the right implementation language there will be a one to one correspondence between the denotational

semantics and the interpreter. We can even say that if we have a denotational semantics we get the interpreter for free. It is no secret that functional programming languages are best suited for such a problem. We do not need imperative features and thus we are free to choose between eager and lazy evaluation. Lazy evalutation has the advantage that e.g. the definition of the fixed point operator is straightforward. For the interpreter we chose the functional programming language Miranda[1] [7]. The language to be interpreted is the object-oriented programming language O'SMALL [5], and thus the T-diagram for the interpreter is:



O'SMALL had originally been developed for semantic specification. It turned out that there seems to be some consensus that O'SMALL contains the essential features of an object-oriented programming language with class inheritance. Gündel [2] wrote a structured operational semantics for it. Palsberg and Schwartzbach [6] independently developed a language almost identical to O'SMALL. This consensus and some interest in getting acquainted with O'SMALL by using it, have encouraged us to make the O'SMALL-interpreter publicly available via ftp. This report tells you where to get the interpreter, and, to a certain extent, how to use it. We also included the Miranda programs for readers who would just like to know how the transformation from the meta language of denotational semantics to a functional programming language is done. We apologize that the language, the system, and this description are in such a prototypical state.

The following sections contain the source code of the interpreter in Miranda, the concrete syntax of O'SMALL, and a pointer to the location where the interpreter and the parser can be retrieved by anonymous ftp.

2 Some comments on the interpreter files

Listings of the interpreter files can be found in section 3. A short introduction to the programming language Miranda can be found in [7]. If you look for a more thorough introduction to functional programming refer to [1]. To understand the interpreter files we suggest that you compare the programs with the definitions and comments of the denotational semantics paper [5]. Miranda is a strongly typed language and thus we must insert type constructors that do not appear in the semantics. Also some of the simplifications to make the semantics more readable must be made explicit.

The O'SMALL-dialect of this interpreter uses explicit wrappers [4]. If you are not interested in explicit wrappers and just want "classical" O'SMALL you may just refer to [5]

[1] Miranda is a trademark of Research Software Ltd..

and the following notice: Instead of writing one class declaration we need a wrapper and a class declaration. In the following O'SMALL-fragments, V represents a sequence of variable declarations and M represents a sequence of method declarations.

`class` A `subclassOf` B `def` V `in` M `ni`

This is *not* accepted by this interpreter! Here, with explicit wrappers, a wrapper W is defined in much the same way as a class before, except that it does not name a superclass.

`wrapper` W `def` V `in` M `ni`

The syntax and semantics of the modification, i.e. V and M, are as before. The class A is defined as the wrapper W applied to the superclass B. The next program line is again O'SMALL-syntax.

`class` A W B

For brevity some of the O'SMALL-examples in the papers use syntactical constructions or primitive functions that are not accepted by this interpreter. For an exact specification consult the concrete syntax in section 4.

The first interpreter file contains the syntactic domains (section 3.1, page 4). Lists are written with brackets in Miranda and so a program consists of a constructor `Program` and a list of wrapper declarations followed by a list of compound expressions. Primitive semantic domains must now be implemented. For technical reasons there are special cases for methods and messages without arguments, and some auxiliary functions.

The second file contains the semantic domains (section 3.2, page 5). The semantic domains are a straightforward translation. Again primitive domains must be implemented.

The third interpreter file contains auxiliary functions and basic definitions (section 3.3, page 6). The first auxiliary function is the generic function $\star$ used for the composition of commands an declarations. In this semantics it appears with two different types, and thus two functions `x1` and `x2` must be declared in Miranda. They are used as infix in the sequel and are thus preceded by a dollar sign. Also for domain checking (D?) we need a collection of functions: one for each domain. The store is implemented as a list, locations as natural numbers. The first three locations in the store are already taken by the error flag, the input, and the output. Some data type transformation functions ...`To`... are needed, because we work in a strongly typed language. This inconvenience is however more than compensated by the advantage of static type inference. Finally there are basic functions for inheritance. The last line is the definition of the fixed point operator. It is just defined by the property that a fixed point `f` must have. This definition would look more complicated in a language with eager evaluation because we would have to insert lambdas to prevent non-termination.

The last file contains the semantic functions (section 3.4, page 10). The semantic functions contain many local declarations of functions called `l` or `l1`. These are necessary because of the absence of a λ-construction in Miranda.

3 The interpreter files

3.1 The syntactic domains

```
ide  == [char]

bas == num

binOp ::= Plus | Minus |
          Mult | Div | Mod |
          Le | Leq | Neq | Gr | Grq | Eq

pro ::= Program [w] [c]

w ::= Wrapper ide [v] [m] | Class ide ide ide

v ::= Var ide expr

m ::= Method ide [ide] [c]

c ::= Expr expr |
      Ass ide expr |
      Outp expr |
      If expr [c] [c] |
      While expr [c] |
      Def v [c]

expr ::= New expr |
         BinOp binOp expr expr |
         Bas bas |
         Sign binOp bas |
         Read |
         Sqrt expr |
         Id ide |
         Message ide ide [expr]


|| Auxiliary functions for nullary methods and messages
method0 ide cL = Method ide [] cL
message0 ide1 ide2 = Message ide1 ide2 []


|| Auxiliary function: in the abstract syntax generated by the parser
```

```
|| generator the empty list cannot be created because the empty expression
|| cannot be parsed. Therefore list with one element are created by the
|| function 'sing'.
sing :: * -> [*]
sing x = [x]


|| Auxiliary function: the concrete syntax allows lists of variables but
|| the abstract syntax allows only one. This function makes the transformation
|| because the parser (created by this generator) cannot do it.
def :: [v] -> [c] -> c
def [v] cList = Def v cList
def (v:rest) cList = Def v [(def rest cList)]


|| Auxiliary function: the parser generates constant names for 'self',
|| 'super', and 'current'. We want to treat 'self', 'super', and 'current'
|| as identifiers, and thus we need strings instead of constants. These
|| are defined here.
self = "self"
super = "super"
current = "current"
```

3.2 The semantic domains

```
%include "synDom.m"

loc == num

bv == num

env == ide -> dvu

file == [rv]

errStop ::= Error | Stop
ans == (file,errStop)


dv ::= Loc loc |
       RvDv rv |
       Meth meth |
       Cl class |
```

```
       Wr wrapper

dvu ::= Dv dv | Unbound

sv ::= File file | RvSv rv

rv ::= Bool bool | Bv bv | Obj env

store == [sv]

meth == [dv] -> store -> (dv,store)

class == stObject -> stObject

stObject == store -> (env,store)

wrapper == stObject -> class
```

3.3 Auxiliary functions

```
%include "synDom.m"
%include "semDom.m"



|| Functions of the basic definitions
||------------------------------------------------------------------------

|| Make the self-distributing version of a binary operator [Hen91c,Def.3.3]
box :: (*->**->***)->(****->*)->(****->**)->****->***
box binaryOp g1 g2 = l where l s = (g1 s) $binaryOp (g2 s)

|| The record modification operator [Hen91c,Def.3.4]
triangle :: class -> stObject -> stObject
triangle w p = (w p) $specialPlus p
               where specialPlus  a b s
                = (aEnv $plus bEnv ,aS)
                   || the store aS contains the instance variables of w and p
                  where
                  (aEnv,aS) = a s
                  (bEnv,bS) = b s

|| [Hen91c,Def.3.2] the left preferential plus operator.
```

```
|| Combines environments such that the first one is preferred.
plus :: env -> env -> env
plus envNew envOld ide = envOld ide, envNew ide  = Unbound
                       = envNew ide, otherwise

fix f = f(fix f)



|| Auxiliary functions (also compare [Gor79,pp50ff,70ff]).
||----------------------------------------------------------------------

|| The star operator for the combination of functions:
|| It is defined in the appendix of "Wrapper Semantics of an
|| Object-Oriented Programming Language with State" LNCS 256 [Hen91c].
|| There are two alternatives for the type of the star operator.
|| For the upper case in the angular brackets in the paper we define
|| the auxiliary function `x1' and for the second `x2'.

x1 :: (store -> (*,store)) -> (* -> store -> (**,store))
                           -> (store -> (**,store))
x1 f g s1 = (undef,s2), s2 $at err = RvSv(Bool True)
          = g d2 s2, otherwise          where (d2,s2) = f s1

x2 :: (* -> store -> (**,store)) -> (** -> store -> (***,store))
                                 -> (* -> store -> (***,store))
x2 f g d1 s1 = (undef,s2), s2 $at err = RvSv(Bool True)
             = g d2 s2, otherwise           where (d2,s2) = f d1 s1



cond :: (*,*) -> dv -> *
cond (d1,d2) (RvDv (Bool True))  = d1
cond (d1,d2) (RvDv (Bool False))  = d2

cont :: dv -> store -> (sv,store)    || no test on validity
cont (Loc l) s = (s $at l,s)


|| Functions corresponding to `D?'
isSv :: dv -> store -> (dv,store)
isSv (RvDv rv) = result (RvDv rv)
isSv dv = seterr
```

```
isDv :: dvu -> store -> (dv,store)
isDv (Dv dv) = result dv
isDv Unbound = seterr

isLoc, isBool,isRv, isObj, isMeth, isCl, isWr :: dv -> store -> (dv,store)
isRv (RvDv rv) = result (RvDv rv)
isRv rv = seterr

isLoc (Loc l) = result (Loc l)
isLoc dv = seterr

isBool (RvDv (Bool True)) = result (RvDv (Bool True))
isBool (RvDv (Bool False)) = result (RvDv (Bool False))
isBool dv = seterr

isObj (RvDv(Obj env)) = result (RvDv(Obj env))
isObj dv = seterr

isMeth (Meth m) = result(Meth m)
isMeth dv = seterr

isCl (Cl class) = result (Cl class)
isCl dv = seterr

isWr (Wr wrapper) = result (Wr wrapper)
isWr dv = seterr
|| End of functions corresponding to ‘D?’



deref :: dv -> store -> (dv,store)
deref (Loc location) s = (svToDv (s $at location),s)
deref (RvDv e) = result (RvDv e)
deref other s
  = error(concat["deref ",show other,show s,"expected storable value"])

new :: store -> (loc,store)
new s = (firstFree s 0, s)
        where
        firstFree [] accLength = accLength
        firstFree (h:t) accLength = firstFree t (accLength + 1)
        || firstFree determines the first free storage cell. There is no
```

```
        || error case, because we assume that the natural numbers, which
        || is the type of locations, are infinite.

result :: * -> store -> (*,store)
result d s = (d,s)

seterr :: store -> (*,store)
seterr s = (undef,upd s (RvSv (Bool True)) err)

update :: loc -> dv -> store -> (dv,store)
update loc = isSv $x2 l
             where
             l (RvDv e) s = ((RvDv e),upd s (RvSv e) loc)




|| Further auxiliary functions and auxiliary function to auxiliary functions
||----------------------------------------------------------------------------

|| The first three locations in the store are already occupied:
err = 0
inp = 1
out = 2

|| ‘at’ searches for a location in the store and returns the contents.
|| The case of the empty store should not occur.
at :: store -> loc -> sv
at (h:t) 0 = h
at (h:t) (l+1) = at t l
at [] = error "’at’: tried to get the contents of unused store"

|| Takes an identifier and a value (type dv) and returns a little
|| environment, where the identifier is bound to that value.
makeEnv :: ide -> dv -> env
makeEnv ide dv = l where l i = Dv dv, i = ide
                             = Unbound ,otherwise

|| upd s e l: puts in s at location l the value e instead of the old value
|| or append e to the store
upd :: store -> sv -> loc -> store
upd (h:t) sv 0 = (sv:t)
upd (h:t) sv (l+1) = h:(upd t sv l)
```

```
upd [] sv 0 = [sv]
upd [] sv n
  = error(concat["upd: ",show sv,show n,"exceeded current extension of store"])

dvuToDv :: dvu -> dv
dvuToDv (Dv dv) = dv
dvuToDv Unbound = error"dvuToDv : identifier was Unbound in environment"

dvToSv :: dv -> sv
dvToSv (RvDv rv) = RvSv rv
dvToSv dv = error(concat["dvToSv ",show dv,"expected RvDv or Obj"])

svToDv :: sv -> dv
svToDv (RvSv rv) = RvDv rv
svToDv (File file)
  = error(concat["svToDv ",show file," cannot convert File to dv"])
```

3.4 The semantic functions

```
%include "synDom.m"
%include "semDom.m"
%include "hilfsFun.m"

|| types of the semantic functions
evP :: pro -> file -> ans
evR, evE :: expr -> env -> store -> (dv,store)
evC :: c -> env -> store -> (dv,store)
evV :: v -> env -> store -> (env,store)
evW :: w -> env -> store -> (env,store)
evM :: m -> env -> env
evO :: binOp -> (rv,rv) -> store -> (dv,store)
evB :: bas -> bv


|| list of the semantic rules
evP (Program wl cl) input
 = extractAns finalStore
   where
   extractAns s = (reverse outp,errSt(s $at err))
   || for efficiency the output is always put at the front of the list, such
   || that here a reversal of the list is necessary.
                where
                errSt (RvSv(Bool True)) = Error
```

```
                  errSt (RvSv(Bool False)) = Stop
                  File outp = s $at out
   (wrClEnv,wrClS) = evWl wl initialEnv initialStore
   (dv,finalStore) = evCl cl wrClEnv initialStore
   initialStore = [RvSv(Bool False), File input, File []]
   initialEnv = makeEnv "Base" (Cl base)
                where base selfRef s = (emptyEnv,s)
                                       where emptyEnv ide = Unbound

evR expr env = ((evE expr env) $x1 deref) $x1 isRv

evE (New expr) env
 = ((evE expr env) $x1 isCl) $x1 makeObj
   where makeObj (Cl c) s = (RvDv(Obj objEnv),newS)
                            where
                            (objEnv,newS) = (fix c) s
evE (BinOp binOp expr1 expr2) env
 = (evR expr1 env) $x1 l1
   where
   l1 (RvDv e1)
    = (evR expr2 env) $x1 l2
      where
      l2 (RvDv e2) = evO binOp (e1,e2)
   l1 other = error(concat["evE Binop  :   expected R-value"])
evE (Bas bas) env = result (RvDv(Bv(evB bas)))    || evB is the identity
evE (Sign binOp bas) env
 = result(RvDv(Bv (res binOp)))
   where
   res Plus = (evB bas)
   res Minus = 0 - (evB bas)
   res op
    = error(concat
           ["evE (Sign ",show binOp,show bas,") :  expected Plus, Minus"])
evE Read env = (cont (Loc inp)) $x1 l
               where l (File []) s = seterr s
                     l (File (h:t)) s = (RvDv h,upd s (File t) inp)
evE (Sqrt expr) env
 = (evE expr env) $x1 sqRoot
   where sqRoot (RvDv(Bv bas)) s = (RvDv(Bv(sqrt bas)),s)
         sqRoot other s = error(concat["evE ",show expr," expected Bv"])
evE (Id ide ) env = (result (env ide)) $x1 isDv
evE (Message objName messSel exprL) env
 = ((evR (Id objName) env) $x1 isObj) $x1 l1
```

```
   where
   l1 (RvDv(Obj objEnv))
    = ((result (dvuToDv(objEnv messSel))) $x1 isMeth) $x1 l2
      where
      l2 (Meth m)
       = (evParams exprL []) $x1 m
         where
         evParams [] resultL s = (reverse resultL,s)
         evParams (p:rest) resultL s = evParams rest (dv:resultL) newS
                                       where
                                       (dv,newS) = (evR p env) s

evC (Expr expr) = evE expr
evC (Ass id expr) env
 = ((evE (Id id) env) $x1 isLoc) $x1 l
   where
   l (Loc location) = (evR expr env) $x1 (update location)
evC (Outp expr) env = (evR expr env) $x1 l
                      where
                      l (RvDv r) s = ((RvDv r),upd s (File (r:outp)) out)
                                     where File outp = (s $at out)
evC (If expr cl1 cl2) env = ((evR expr env) $x1 isBool) $x1
                            cond(evCl cl1 env,evCl cl2 env)
evC (While expr cl) env
 = ((evR expr env) $x1 isBool) $x1
   cond((evCl cl env) $x1 l,result(RvDv(Bool False)))
   where
   l e = evC (While expr cl) env

evC (Def v cl) env = (evV v env) $x1 l
                     where l newEnv = evCl cl (newEnv $plus env)

evCl :: [c] -> env -> store -> (dv,store)
evCl (c:[]) = evC c
evCl (c:rest) env = (evC c env) $x1 l
                    where
                    l e = evCl rest env

evV (Var iden expr) env
 = (evR expr env) $x1 l
   where
   l e = new $x1 l1
         where
```

```
        l1 location s
         = (makeEnv iden (Loc location),upd s (dvToSv e) location)

evW (Class className wrapperName parentName)env
 = ((evE (Id parentName) env) $x1 isCl) $x1 l1
   where
   l1 (Cl parent)
    = ((evE(Id wrapperName) env) $x1 isWr) $x1 l2
      where
      l2 (Wr wrapper)
       = result(makeEnv className(Cl ((box triangle) wrapper parent)))
evW (Wrapper wrapperName vl ml) env
 = result(makeEnv wrapperName(Wr wrapper))
   where
   wrapper self super sCreate
    = (evMl ml (((makeEnv "self" (RvDv(Obj selfEnv)))$plus
      (makeEnv "super" (RvDv(Obj superEnv)))) $plus
      (localEnv $plus (makeEnv "current" (Cl cur)) $plus env)),newS)
      || By combining the three environments we obtain encapsulation.
      || The instance variables of the ancestor classes are invisible.
      || They cannot be reached via ‘super’ because they are no methods.
      where cur x = self
            (localEnv,newS) = evVl vl env pS
            (selfEnv,sS) = self sCreate    ||store at object creation
            (superEnv,pS) = super sCreate  ||store at object creation
|| The store for the instance variable of the superclass is allocated by
|| ‘super’. The store for the new instance variables is allocated during
|| the evaluation of vl. We account for the instance variables of the
|| superclass by starting from ‘pS’; the changed store is called ‘newS’.
||
|| ‘self’ must be supplied with ‘sCreate’ in order to point to the right
|| instance variables. These instance variables are in ‘localEnv’ and
|| in the local environments of the superclasses.
|| ‘self’ is recursive and lazy evaluation is important here.
|| If one looked at ‘sS’ one would enforce complete evaluation of
|| ‘self sCreate’. This would lead to non-termination.



|| variable and class definitions are not mutually recursive
evVl :: [v] -> env -> store -> (env,store)
evVl [] env = result env
evVl (v:rest) env = (evV v env) $x1 l1
```

```
                    where l1 r1 = (evVl rest (r1 $plus env))
evWl :: [w] -> env -> store -> (env,store)
evWl [] env = result env
evWl (w:rest) env = (evW w env) $x1 l1
                    where l1 r1 = (evWl rest (r1 $plus env))

|| method definitions are not mutually recursive
|| because all recursion goes via ‘self’
evM(Method methName parNameL cl) env
 = makeEnv methName (Meth body)
   where body parL
    = evCl cl (methEnv parNameL parL)
      where
      methEnv [] [] = env
      methEnv (parName:pNL)(par:pL) = (makeEnv parName par)
                                      $plus (methEnv pNL pL)
      methEnv pNL pL
       = error(concat
               ["number of actual and formal parameters differs in: ",
               show methName])

evMl :: [m] -> env -> env
evMl [] env = env
evMl (m:rest) env = (evM m env) $plus (evMl rest env)


evO Plus (Bv n1,Bv n2)  = result(RvDv(Bv(n1+n2)))
evO Minus (Bv n1,Bv n2) = result(RvDv(Bv(n1-n2)))
evO Mult (Bv n1,Bv n2)  = result(RvDv(Bv(n1*n2)))
evO Div (Bv n1,Bv n2)   = result(RvDv(Bv(n1 div n2)))
evO Mod (Bv n1,Bv n2)   = result(RvDv(Bv(n1 mod n2)))
evO Le (Bv n1,Bv n2)    = result(RvDv(Bool(n1 < n2)))
evO Leq (Bv n1,Bv n2)   = result(RvDv(Bool(n1 <= n2)))
evO Neq (Bv n1,Bv n2)   = result(RvDv(Bool(n1 ~= n2)))
evO Gr (Bv n1,Bv n2)    = result(RvDv(Bool(n1 > n2)))
evO Grq (Bv n1,Bv n2)   = result(RvDv(Bool(n1 >= n2)))
evO Eq (Bv n1,Bv n2)    = result(RvDv(Bool(n1 = n2)))

evB = id
```

4 The concrete syntax of O'SMALL

The O'SMALL-version here contains explicit wrappers. The parser has been generated with an ELL(2)-parser generator by Heckmann [3]. Letters are a,b,c,...,z and A,B,C,...,Z. Digits are 0,1,2,...,9. An identifier (iden) is a letter followed by letters or digits. A number is at least one digit. Operators are divided into three classes:

mulop = * | / | %
addop = + | −
relop = <|<=|<>|>|>=|=

They stand for the following functions:

operator	type	function
*	num × num → num	multiplication
/	num × num → num	whole numbered division (div)
%	num × num → num	modulo (mod)
+	num × num → num	addition
−	num × num → num	subtraction
<	num × num → bool	less
<=	num × num → bool	less or equal
<>	num × num → bool	not equal
>	num × num → bool	greater
>=	num × num → bool	greater or equal
=	num × num → bool	equal

The data type num consists of integers and floating-point numbers. There are the following terminal symbols: **read, output, if, then, else, fi, while, do, od, def, in, ni, var, meth, program, class, wrapper, sqrt, new, self, super, current.** Comments can either be enclosed in braces ({...}) or in parentheses with a star ((*...*)). Nonterminals are written with capital letters. Braces ({}), brackets ([]) and bars (|) are used as meta symbols and denote optional elements, parentheses and alternatives respectively. The starting symbol (axiom) is P. Keywords are written in bold typeface. The productions are:

P	:=	WL CL { ; | . }
W	:=	**wrapper** iden **def** VL **in** ML **ni** | **class** iden iden iden
V	:=	**var** iden := E
M	:=	**meth** iden ({PL}) CL
C	:=	E | iden := E | **output** E | **if** E **then** CL **else** CL **fi** | **while** E **do** CL **od** | **def** VL **in** CL **ni**
E	:=	S | S relop S | **new** E
S	:=	T | T addop S
T	:=	B | B mulop T
B	:=	num | addop num | **sqrt** (E) | **read** | (E) | **current** | [**self** | iden] { . iden { (AL) } } | **super** . iden { (AL) } }

CL, VL, WL, and ML are list constructions of C, V, W, and M respectively. PL are lists of identifiers (iden). AL are lists of expressions (E). C are separated by semicolons (;). V, W, and M may be separated by semicolons. Parameter lists and argument lists are separated by commas.

Note that for some technical reasons (we have an LL-parser) the receiver of a message must be a variable and cannot be a general expression.

Sometimes an example is better than many definitions. An example of a program that is accepted by this parser is contained in figure 1. This program demonstrates the necessity to program around some of the missing primitive functions and also shows the use of wrappers instead of classes that was mentioned in section 2.

5 FTP distribution

To obtain the compiler by internet ftp, connect to host ftp.cs.uni-sb.de use login id "anonymous" with your name as password. You will be in the directory `/pub`. Go to directory `/pub/osmall` ("cd osmall"). Then put ftp in binary mode ("binary") and "get" the relevant files in that directory.

Host:	Net Address:	Login:	Passwd:	Directory:
ftp.cs.uni-sb.de	134.96.7.254	anonymous	Your name	/pub/osmall

```
wrapper PointWrap
def var xComp := 0  var yComp := 0
in meth x() xComp
   meth y() yComp
   meth move(X,Y) xComp := X+xComp; yComp := Y+yComp
   meth distFromOrg()  sqrt(xComp*xComp + yComp*yComp)
   meth closerToOrg(point) self.distFromOrg < point.distFromOrg
ni
class Point PointWrap Base

wrapper CircleWrap
def var radius := 0
in meth r() radius
   meth setR(r) radius := r
   meth distFromOrg()
      def var d := super.distFromOrg - self.r
      in if d > 0 then d else 0 fi ni
ni
class Circle CircleWrap Point

def var p := new Point
    var c := new Circle
in p.move(2,2); c.move(3,3); c.setR(2);
   output p.closerToOrg(c);
   p.move(0,-2);    c.move(0,-2);
   output p.closerToOrg(c)
ni                                {result:  false, false}
```

Figure 1: O'SMALL program with points and circles in concrete syntax

The directory /pub/osmall contains a file named `README` that contains this information and the compressed tar file `osmall.0.1.tar.Z`. It contains a directory called `parser` and a directory called `interpreter`.

NOTE: Ftp should be put into binary mode before transferring the compressed tar file.

Here is a sample dialog:

```
ftp
ftp> open ftp.cs.uni-sb.de
Name: anonymous
Password: <your name>
ftp> binary
```

```
ftp> cd osmall
ftp> get README
ftp> get osmall.0.1.tar.Z
ftp> close
ftp> quit
```

After the files are transferred they should be uncompressed using the uncompress command and then extracted using tar into a directory called `osmall`. For example:

```
mkdir osmall
mv osmall.0.1.tar.Z osmall
cd osmall
uncompress -c osmall.0.1.tar.Z | tar xf -
```

will unpack the directories.

6 Running an O'SMALL program

Go into the directory parser and type
`make`
An executable file called `osmall` will be the result. `osmall` is the parser. It is best called with
`osmall < 'file1' > 'file2'`
where `file1` contains the O'SMALL program and `file2` will contain the abstract syntax of this program. To use the program in abstract syntax in the Miranda-system it must be declared as a value. Before it is declared the syntactic domains must be known. To run the program the semantic functions, which rely on the semantic domains and the auxiliary functions, must be known. The easiest way to run the program is to create a file called `ex.m` with the contents:

```
%include "synDom.m"
%include "semDom.m"
%include "hilfsFun.m"
%include "semFun.m"
ex = 'file2'
```

where 'file2' is the program in abstract syntax where you have removed the leading string "ip". Move the file `ex.m` to the directory `interpreter`, where the files `synDom.m, semDom.m, hilfFun.m, semFun.m` are. Now call
`mira ex`
in the directory `interpreter`. You will get the Miranda prompt after the system has compiled everything. Then you may type

```
evP ex []
```

in the Miranda system if the input is empty. Otherwise you will write some input between the brackets.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1988.

[2] A. Gündel. Objektorientierte Programmiersprachen. lecture notes by Gerd Lierhaus, Universtät Dortmund, 1990.

[3] R. Heckmann. Manual for the ELL(2)-parser generator and tree generator generator. Technical Report Doc.: S.1.1-R-2.1, European Strategic Programme for Research and development in Information Technology, Aug. 1986. PROSPECTRA, Project Ref. No. 390.

[4] A. V. Hense. Denotational semantics of an object-oriented programming language with explicit wrappers. Technical Report A 11/90, Universität des Saarlandes, Fachbereich 14, June 1990.

[5] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568. Springer-Verlag, Sept. 1991.

[6] J. Palsberg and M. Schwartzbach. Object-oriented type inference. Technical Report DAIMI PB-345, Aarhus University, Mar. 1991.

[7] D. Turner. Miranda: A non-strict functional language with polymorphic types. *Lecture Notes in Computer Science*, 201:1–16, 1985. Functional Programming Languages and Computer Architecture.