Semantics of Pointers[*]

Hanne Riis Nielson

Informatics and Mathematical Modelling
Technical University of Denmark

February 26, 2003

1 Syntax of a pointer language

We shall study an extension of WHILE that allows us to create cells in the heap; the cells are structured and may contain values as well as pointers to other cells. The data stored in a cell is accessed via selectors so we assume that a finite and non-empty set **Sel** of *selector names* are given:

$$sel \in \mathbf{Sel} \qquad \text{selector names}$$

As an example **Sel** may include the Lisp-like selectors `car` and `cdr` for selecting the first and second components of pairs. The cells of the heap can be addressed by expressions like `x.cdr`: this will first determine the cell pointed to by the variable `x` and then return the value of the `cdr` field. For the sake of simplicity we shall only allow one level of selectors although the development generalises to several levels. Formally the *pointer expressions*

$$p \;\in\; \mathbf{PExp}$$

are given by:

$$p \;::=\; x \mid x.sel$$

The syntax of the WHILE-language is now extended to have:

$$\begin{array}{rcl}
a & ::= & p \mid n \mid a_1\ op_a\ a_2 \mid \texttt{nil} \\
b & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid op_p\ p \\
S & ::= & p\texttt{:=}a \mid \texttt{skip} \mid S_1;\ S_2 \mid \\
 & & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \\
 & & \texttt{malloc}\ p
\end{array}$$

[*]This note is based on Section 2.6 of "Principles of Program Analysis" (by F. Nielson, H. Riis Nielson and C. Hankin) published by Springer 1999.

Arithmetic expressions are extended to use pointer expressions rather than just variables, and an arithmetic expression can also be the constant `nil`. The binary operations op_a are as before, that is, they are the standard arithmetic operations and in particular they do *not* allow pointer arithmetic. The boolean expressions are extended such that the relational operators op_r now allow testing for the *equality of pointers* and also we shall allow unary operations op_p on pointers (as for example `is-nil` and `has-`*sel* for each $sel \in \mathbf{Sel}$). Note that arithmetic as well as boolean expressions can only access cells in the heap, they cannot create new cells nor update existing cells.

The assignment statement takes the general form p`:=`a where p is a pointer expression. In the case where p is just a variable we have an extension of the ordinary assignment of the WHILE language and in the case where p contains a selector we have a destructive update of the heap. The statements of the extended language also contain a statement `malloc` p for creating a new cell pointed to by p.

Example The following program reverses the list pointed to by `x` and leaves the result in `y`:

```
y:=nil;
while not is-nil(x) do
      (z:=y; y:=x; x:=x.cdr; y.cdr:=z);
z:=nil
```

Figure 1 illustrates the effect of the program when `x` points to a five element list and `y` and `z` are initially undefined. Row 0 shows the heap just before entering the `while`-loop: `x` points to the list and `y` is `nil` (denoted by $\diamond$); to avoid cluttering the figure we do not draw the `car`-pointers. After having executed the statements of the body of the loop the situation is as in row 1: `x` now points to the tail of the list, `y` points to the head of the list and `z` is `nil`. In general the n'th row illustrates the situation just before entering the loop the $n+1$'th time so in row 5 we see that `x` points to `nil` and the execution of the loop terminates and `y` points to the reversed list. The final statement `z:=nil` simply removes the pointer from `z` to ξ_4 and sets it to the `nil`-value. □

2 Structural Operational Semantics

To model the scenario described above we shall introduce an infinite set **Loc** of *locations* (or addresses) for the heap cells:

$$\xi \in \mathbf{Loc} \qquad \text{locations}$$

The value of a variable will now either be an integer (as before), a location (i.e. a pointer) or the special constant $\diamond$ reflecting that it is the `nil` value. Thus the

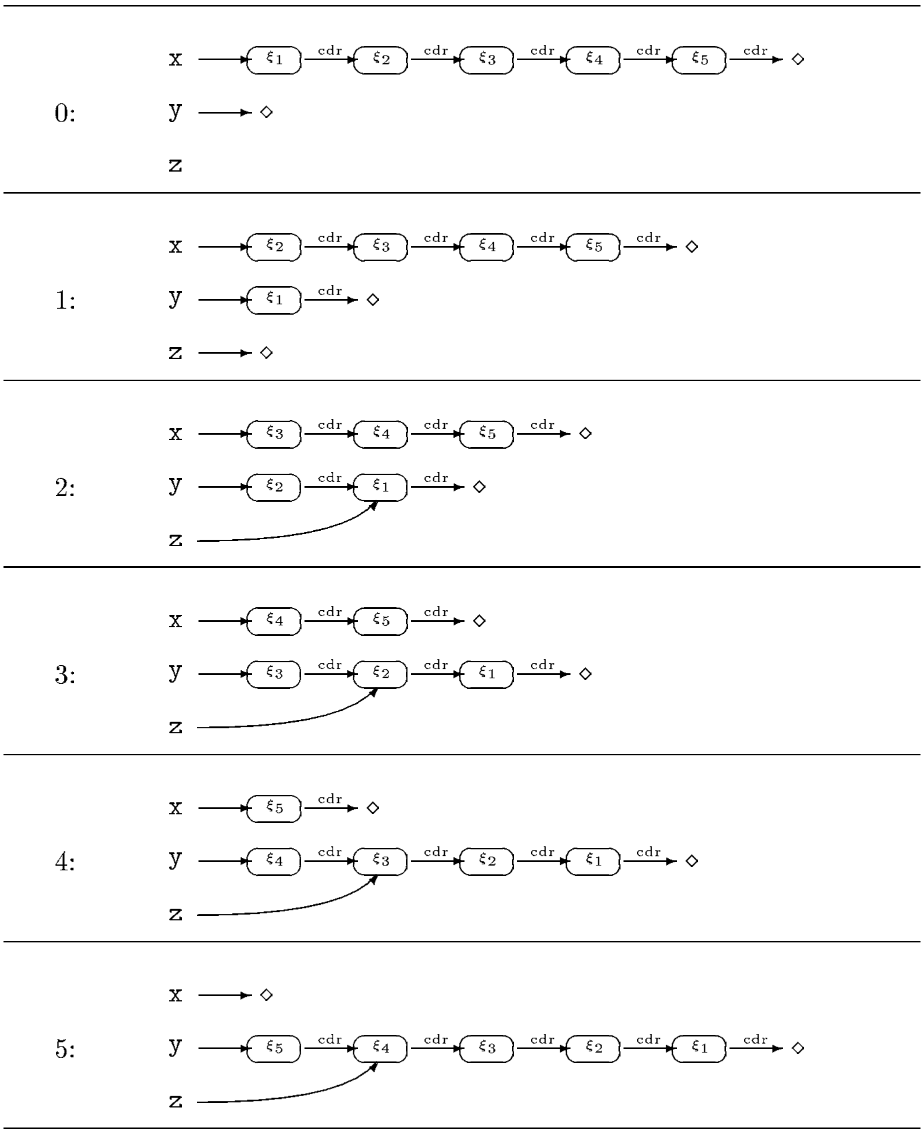
0:
x
ξ_1 cdr ξ_2 cdr ξ_3 cdr ξ_4 cdr ξ_5 cdr $\diamond$
y
$\diamond$
z
1:
x
ξ_2 cdr ξ_3 cdr ξ_4 cdr ξ_5 cdr $\diamond$
y
ξ_1 cdr $\diamond$
z
$\diamond$
2:
x
ξ_3 cdr ξ_4 cdr ξ_5 cdr $\diamond$
y
ξ_2 cdr ξ_1 cdr $\diamond$
z
3:
x
ξ_4 cdr ξ_5 cdr $\diamond$
y
ξ_3 cdr ξ_2 cdr ξ_1 cdr $\diamond$
z
4:
x
ξ_5 cdr $\diamond$
y
ξ_4 cdr ξ_3 cdr ξ_2 cdr ξ_1 cdr $\diamond$
z
5:
x
$\diamond$
y
ξ_5 cdr ξ_4 cdr ξ_3 cdr ξ_2 cdr ξ_1 cdr $\diamond$
z

Figure 1: Reversal of a list of five elements.

states are given by

$$\sigma \in \mathbf{State} = \mathbf{Var} \hookrightarrow (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

where **Var** is the set of variables occurring in the program of interest. As mentioned above the cells of the *heap* have multiple fields and they are accessed using the selectors. Each field can either be an integer, a pointer to another cell or it can be `nil`. We formalise this by taking

$$\mathcal{H} \in \mathbf{Heap} = (\mathbf{Loc} \times \mathbf{Sel}) \hookrightarrow (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

where the use of *partial* functions with finite domain reflects that not all selector fields need to be defined; as we shall see later, a newly created cell with location ξ will have all its fields to be uninitialised and hence the corresponding heap $\mathcal{H}$ will have $\mathcal{H}(\xi, \mathit{sel})$ to be undefined for all $\mathit{sel} \in \mathbf{Sel}$.

Pointer expressions. Given a state and a heap we need to determine the value of a pointer expression p as an element of $\mathbf{Z} + \mathbf{Loc} + \{\diamond\}$. For this we introduce the function

$$\wp \; : \; \mathbf{PExp} \to (\mathbf{State} \times \mathbf{Heap}) \hookrightarrow (\mathbf{Z} + \{\diamond\} + \mathbf{Loc})$$

where **PExp** denotes pointer expressions with variables in **Var**. It is defined by:

$$\begin{array}{rcl} \wp[\![x]\!](\sigma, \mathcal{H}) & = & \sigma(x) \\ \wp[\![x.\mathit{sel}]\!](\sigma, \mathcal{H}) & = & \left\{ \begin{array}{l} \mathcal{H}(\sigma(x), \mathit{sel}) \\ \quad \text{if } \sigma(x) \in \mathbf{Loc} \text{ and } \mathcal{H} \text{ is defined on } (\sigma(x), \mathit{sel}) \\ \mathit{undef} \\ \quad \text{if } \sigma(x) \notin \mathbf{Loc} \text{ or } \mathcal{H} \text{ is undefined on } (\sigma(x), \mathit{sel}) \end{array} \right. \end{array}$$

The first clause takes care of the situation where p is a simple variable and using the state we determine its value – note that this may be an integer, a location or the special `nil`-value $\diamond$. The second clause takes care of the case where the pointer expression has the form $x.\mathit{sel}$. Here we first have to determine the value of x; it only makes sense to inspect the *sel*-field in the case x evaluates to a location that has a *sel*-field and hence the clause is split into two cases. In the case where x evaluates to a location we simply inspect the heap $\mathcal{H}$ to determine the value of the *sel*-field – again we may note that this can be an integer, a location or the special value $\diamond$.

Example In Figure 1 the *oval nodes* model the cells of the heap $\mathcal{H}$ and they are labelled with their location (or address). The *unlabelled edges* denote the state σ: an edge from a variable x to some node labelled ξ means that $\sigma(x) = \xi$; an edge from x to the symbol $\diamond$ means that $\sigma(x) = \diamond$. The *labelled edges* model the heap $\mathcal{H}$: an edge labelled *sel* from a node labelled ξ to a node labelled ξ'

means that there is a *sel* pointer between the two cells, that is $\mathcal{H}(\xi, sel) = \xi'$; an edge labelled *sel* from a node labelled ξ to the symbol $\diamond$ means that the pointer is a `nil`-pointer, that is $\mathcal{H}(\xi, sel) = \diamond$.

Consider the pointer expression `x.cdr` and assume that σ and $\mathcal{H}$ are as in row 0 of Figure 1, that is $\sigma(\texttt{x}) = \xi_1$ and $\mathcal{H}(\xi_1, \texttt{cdr}) = \xi_2$. Then $\wp[\![\texttt{x.cdr}]\!](\sigma, \mathcal{H}) \ = \ \xi_2$.

Arithmetic and boolean expressions. It is now straightforward to extend the semantics of arithmetic and boolean expressions to handle pointer expressions and the `nil`-constant. Obviously the functionality of the semantic functions $\mathcal{A}$ and $\mathcal{B}$ has to be changed to take the heap into account:

$$\begin{array}{rcl}\mathcal{A} & : & \textbf{AExp} \to (\textbf{State} \times \textbf{Heap}) \hookrightarrow (\mathbf{Z} + \textbf{Loc} + \{\diamond\}) \\ \mathcal{B} & : & \textbf{BExp} \to (\textbf{State} \times \textbf{Heap}) \hookrightarrow \mathbf{T}\end{array}$$

The clauses for arithmetic expressions are

$$\begin{array}{rcl}\mathcal{A}[\![p]\!](\sigma, \mathcal{H}) & = & \wp[\![p]\!](\sigma, \mathcal{H}) \\ \mathcal{A}[\![n]\!](\sigma, \mathcal{H}) & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![a_1 \ op_a \ a_2]\!](\sigma, \mathcal{H}) & = & \mathcal{A}[\![a_1]\!](\sigma, \mathcal{H}) \ \mathbf{op}_a \ \mathcal{A}[\![a_2]\!](\sigma, \mathcal{H}) \\ \mathcal{A}[\![\texttt{nil}]\!](\sigma, \mathcal{H}) & = & \diamond\end{array}$$

where we use $\wp$ to determine the value of pointer expressions and we explicitly write that the meaning of `nil` is $\diamond$. Also the meaning $\mathbf{op}_a$ of the binary operation op_a has to be suitably modified to be undefined unless both arguments are integers in which case the results are as for the WHILE-language.

The definition of the semantics of boolean expressions is similar so we only give two of the clauses:

$$\begin{array}{rcl}\mathcal{B}[\![a_1 \ op_r \ a_2]\!](\sigma, \mathcal{H}) & = & \mathcal{A}[\![a_1]\!](\sigma, \mathcal{H}) \ \mathbf{op}_r \ \mathcal{A}[\![a_2]\!](\sigma, \mathcal{H}) \\ \mathcal{B}[\![op_p \ p]\!](\sigma, \mathcal{H}) & = & \mathbf{op}_p \ (\wp[\![p]\!](\sigma, \mathcal{H}))\end{array}$$

Analogously to above, the meaning $\mathbf{op}_r$ of the binary relation operator op_r has to be suitably modified to give undefined in case the arguments are not both integers or both pointers (in which case the equality operation tests for the equality of the pointers). The meaning of the unary operation op_p is defined by $\mathbf{op}_p$; as an example:

$$\textbf{is-nil}(v) \ = \ \begin{cases} \mathsf{tt} & \text{if } v = \diamond \\ \mathsf{ff} & \text{otherwise} \end{cases}$$

Statements. Finally, the semantics of statements is extended to cope with the heap component. The configurations will now contain a state as well as a heap so they have the form

$$\langle S, \sigma, \mathcal{H} \rangle$$

$[ass_1]$	$\langle x\texttt{:=}a, \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma,\mathcal{H})], \mathcal{H}\rangle$	if $\mathcal{A}[\![a]\!](\sigma,\mathcal{H})$ is defined
$[ass_2]$	$\langle x.sel\texttt{:=}a, \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \mathcal{A}[\![a]\!](\sigma,\mathcal{H})]\rangle$	if $\sigma(x) \in \mathbf{Loc}$ and $\mathcal{A}[\![a]\!](\sigma,\mathcal{H})$ is defined
$[skip]$	$\langle \texttt{skip}, \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma, \mathcal{H}\rangle$	
$[seq_1]$	$\dfrac{\langle S_1, \sigma, \mathcal{H}\rangle \Rightarrow \langle S_1', \sigma', \mathcal{H}'\rangle}{\langle S_1; S_2, \sigma, \mathcal{H}\rangle \Rightarrow \langle S_1'; S_2, \sigma', \mathcal{H}'\rangle}$	
$[seq_2]$	$\dfrac{\langle S_1, \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma', \mathcal{H}'\rangle}{\langle S_1; S_2, \sigma, \mathcal{H}\rangle \Rightarrow \langle S_2, \sigma', \mathcal{H}'\rangle}$	
$[if_1]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma, \mathcal{H}\rangle \Rightarrow \langle S_1, \sigma, \mathcal{H}\rangle$	if $\mathcal{B}[\![b]\!](\sigma,\mathcal{H}) = \mathit{true}$
$[if_2]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma, \mathcal{H}\rangle \Rightarrow \langle S_2, \sigma, \mathcal{H}\rangle$	if $\mathcal{B}[\![b]\!](\sigma,\mathcal{H}) = \mathit{false}$
$[wh_1]$	$\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma, \mathcal{H}\rangle \Rightarrow \langle (S; \texttt{while}\ b\ \texttt{do}\ S), \sigma, \mathcal{H}\rangle$	if $\mathcal{B}[\![b]\!](\sigma,\mathcal{H}) = \mathit{true}$
$[wh_2]$	$\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma, \mathcal{H}\rangle$	if $\mathcal{B}[\![b]\!](\sigma,\mathcal{H}) = \mathit{false}$
$[mal_1]$	$\langle \texttt{malloc}\ x, \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma[x \mapsto \xi], \mathcal{H}\rangle$ where ξ does not occur in σ or $\mathcal{H}$	
$[mal_2]$	$\langle \texttt{malloc}\ (x.sel), \sigma, \mathcal{H}\rangle \Rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \xi]\rangle$ where ξ does not occur in σ or $\mathcal{H}$ and	$\sigma(x) \in \mathbf{Loc}$

Table 1: The Structural Operational Semantics of WHILE with pointers.

The clause $[ass_1]$ reflects that for the assignment $x\texttt{:=}a$ the state is updated as usual and the heap is left unchanged. In the case where we assign to a pointer expression containing a selector field we shall leave the state unchanged and update the heap as shown in the clause $[ass_2]$. Here the side condition ensures that the left hand side of the assignment does indeed evaluate to a location.

The construct `malloc` p is responsible for creating a new cell. We have two clauses depending on the form of p. In both cases we introduce a fresh location ξ but we do not specify any values for $\mathcal{H}(\xi, sel)$ – as discussed before we have settled for a semantics where the fields of ξ are undefined; obviously other choices are possible. Also note that in the clause $[mal_2]$ the side condition ensures that we already have a location corresponding to x and hence can create an edge to the new location.

Remark. The semantics only allows a limited reuse of garbage locations. For a statement like

```
malloc x; x:=nil; malloc y
```

we will assign some location to `x` at the first statement and since it neither occurs in the state nor the heap after the second assignment we are free to reuse it in the third statement (but we do not have to). For a statement like

```
malloc x; x.cdr:=nil; x:=nil; malloc y
```

we would not be able to reuse the location allocated in the first statement although it will be unreachable (and hence garbage) after the third statement.