New implementation of sets in New Stanford Pascal

First of all: please excuse possible errors in my English; I am German and not a native English speaker … I will do the best I can.

A Moshix video of 2023 tells about the successful installation of New Stanford Pascal on Hercules, look here: https://www.youtube.com/watch?v=li2J4K2JwtE

But when Moshix tried to write and compile his 8-queens program, it failed.

As it turned out, the reason was that the program used a set with a base type containing negative integers like the following:

```
program QUEENS ( OUTPUT ) ;

const CMIN = 1 ;
      CMAX = 8 ;
      DMIN = - 7 ;
      DMAX = 7 ;

type CRANGE = CMIN .. CMAX ;
     COLUMN = set of CRANGE ;
     FIELD = array [ CRANGE ] of CRANGE ;
     DRANGE = DMIN .. DMAX ;
     DIAG = set of DRANGE ;
```

see the type DIAG, which is in fact a set of -7 .. 7

(I think the type DIAG is used to record which diagonale of the chess board is covered by the queens already on the board …)

The Compiler of 2023, as it turned out, didn‘t support sets with negative integers. It didn‘t complain, but silently generated wrong code.

That‘s why I had to invest some time and energy in finding a new solution for sets.

Let me first describe, how sets were implemented prior to 2024.

Sets before 2024

I already did some improvements to sets, but I did not address the basic shortcomings. In Stanford Pascal, the sets before 2024 are all stored starting with the element zero, no matter if they are supposed to have negative elements (they are simply ignored) or if they start at a higher element, as in set of 10 .. 20.

A set is implemented as a bit list, that is, there is a bit for every element in the set.

The leftmost bit of the set is the element zero (if the base type is integer) or the first value of a scalar type (which is internally represented as zero). The next bit is one, the following two and so on.

In 2023, the sets were limited to 2000 bits, which makes ca. 250 bytes.

The sets in storage had no length informations; the length informations are part of the set instructions. We have SLD (set load), which loads a set from storage to the stack (that is: builds a set descriptor, see below), and then SMV (set move), which assigns sets, even of different lengths. And we have UNI for union of sets, INT for intersection and so on …

When sets are processed, a sort of „set descriptor“ is transferred to the stack, which consists of a 3 byte address of the set (in storage) and a 1 byte set length (in bytes). This 1 byte length limits the sets to 2000 elements, and the 3 byte address is a problem, BTW, for the upcoming AMODE 31 support of the mainframe variants of New Stanford Pascal.

So there is enough motivation to design a „new“ implementation of New Stanford Pascal sets.

New Sets of 2024

The sets must support negative starting points (and positive starting points, which are not zero). And: the length information should not be part of the set descriptors any more, so that there is no 3 byte limit for the addresses.

Otherwise, if the length information stayed in the descriptors, they would have to be expanded to 8 bytes, but this would have a much bigger overall impact on the compiler etc.

The soundest solution IMO is:

The set representation in storage gets 4 more bytes (at the beginning); the first two bytes is the net length of the set representation (number of bytes after the 4 byte prefix). The next two bytes contain the set origin, that is, the internal value of the first bit of the set representation. This should always be a multiple of 8 (and: can be negative, of course).

Two examples:

a) set of 10 .. 30

   this will start at element 8 internally and will need 3 bytes – in fact 7 because of the 4 byte prefix (elements 8 thru 31)

   0003 0008 xx xx xx

b) set of -7 .. 7 (from the chess example)

   starts at -8 and needs only two bytes (in fact 6)

   0002 FFF8 xx xx

An optimization comes to mind:

because all set origins are multiples of 8, it is possible to record the set origin divided by 8 in the second field of the prefix. This way base types in the range -262.136 to +262.136 are possible. Using only positive lengths for the net length, the difference max – min may not exceed 262.136. We‘ll start with that; that‘s a hundredfold increase to what we had in 2023.

So the two examples above will be changed like follows:

0003 0001 xx xx xx

0002 FFFF xx xx

Smooth migration

The problem now is:

how can I change the set implementation, when the sets are heavily used inside the compiler itself? If I change the implementation of the sets, the compiler will not be able to compile itself.

The only solution to this problem IMO is: the new implementation of sets must rely completely on new P-Code instructions, which are different from the existing instructions. The old P-Code instructions must work the same way as before, so that the compiler can continue to use them, and the P-Code interpreter (or 370-translator) can be extended to support the new P-Codes, without discontinuing support for the old P-Codes.

The existing compiler works with the existing instructions and is used in the first stage to compile test programs only. Then these test programs (like the Moshix 8-queens program, for example) are tested, until they work correctly.

And then (stage 2), the new compiler is used to compile itself.

If this works – with the new set implementation – the old P-Code instructions can be phased out.

To do this, I have to identify in the first place, which P-Code instructions need changes.

The first change is related to the definitions of sets, that is: the calculation of the lengths of set types and variables and the definition of set constants in P-Code.

This is an example of two set constants in P-Code today:

```
16 DFC S,X16'00818181818181818181818181818181'
32 DFC S,X28'00818181818181818181818181818181818181818181818181',
     '8181818181'
```

The set constants (and variables) have been defined like this:

```
type SETX = set of - 100 .. 100 ;
     SETY = set of 100 .. 200 ;

var X : SETX := [ ] ;
    Y : SETY := [ ] ;
```

This is an error, because the constant should contain all hex zeroes. Furthermore, the first constant contains only 128 bits (16 * 8) and the second one 224 (28 * 8) – you see the problem with the zero origin. The compiler ignores the -100 origin from the first set type and the +100 origin from the second one completely and starts both sets at origin zero.

The compiler (of 2023) reserves 16 bytes for X and 28 bytes for Y.

The set constants, as shown, are the initialization patterns; the initializations must be carried out at run time (the variables are automatic variables).

The instructions which do this look like this:

```
 LOC 39
 VPU 1,452
 LCA P,$PASMAI#
 INC A,16
 LDA 1,408
 MOV -44
```

This is interesting, because a simple block move is used to do the initialization for both variables, which means that this mechanism should also work later for the new set implementation.

Now, how should the definition of the variables etc. look today with the new compiler?

Given these definitions:

```
type SETX = set of - 100 .. 100 ;
     SETY = set of 100 .. 200 ;

var X : SETX := [ ] ;
    Y : SETY := [ ] ;
```

the new set implementation should do the following:

- reserve a bit string from -104 (next lower multiple of 8) to +100 for SETX, which means 205 bits, which means 26 bytes – plus 4 for the prefix, which makes 30 bytes.
- reserve a bit string from 96 to 200, which means 18 bytes (including the 4 bytes prefix)
- built an initialization pattern for X and Y in the constant area, which allows a simple block move during the procedure startup phase, that is:

```
16 DFC S26,-104,X'0‘
46 DFC S14,96,X‘0‘
```

This is a new syntax for set constants – the number after the S is the length of the sets (stored in the first two bytes of the prefix), then the internal value of the leftmost bit (which is stored divided by 8) and then the bit string in hex notation, which is padded to the right by zeroes, up to the length from above.

Using this notation, it would be possible to support both notations (the old one and the new one) at the same time by P-Code interpreters and translators (to support a smooth migration).

Please note that the internal representation of such „new“ set constants depends on the platform (because of the numbers in the prefix, which are subject to endianness issues). What I show here, therefore, is the mainframe representation:

```
000010: 001A FFF3 000000 ...
00002E: 000E 000C 000000 …
```

Note:

X‘FFFE‘ = -13 (dec.) which means -104

X‘000C‘ = 12 (dec.) which means +96

because to get the real set origin, you have to multiply the number stored by 8.

I have to add one another important remark:

To support the initialization above (by block move), the patterns in the constant area (which are empty sets, after all) have to be created twice in different format, because of the different sizes and origins of the two set types.

But …

in normal operation, when assigning empty sets to set variables, we can use another simple representation of the empty set, which is invariant of the target set type.

This representation is:

0000 0000

That is:

length of the bit string is zero, and the set origin is zero, too.

When assigning this set to a set variable, we have to create a NEW set instruction (which we have to do anyway), which respects the different lengths of source and target AND the possible different set origins.

Let‘s assume, that we assign an empty set (which looks like the pattern above) to a set of type SETY from above, that is:

```
type SETY = set of 100 .. 200 ;

var Y : SETY ;

...

 Y := [ ] ;
```

In this case, the origin of the source set (which is zero) is OUTSIDE the set margins of the set type SETY … but IMO, this doesn‘t matter as long as there are no set elements in the source set which need to be transferred and which would be outside of the margins of the target set.

Same goes for a source set which has LARGER margins than a target set. An assignment IMO is valid, as long as all elements of the source set fit into the target set. (Maybe a runtime check could be inserted, depending on a compiler option … we‘ll see later).

So, after comtemplating some time about the layout of sets in storage and about how new set constant definitions should look, I already changed the compiler to create set constant definitions (in P-Code) like designed above, and now it comes to the next step.

Defining new P-Code instructions

The existing P-Codes which deal with set constants are DFC and LCA.

They can both be kept, IMO, because the new set constants can be distinguished from the old ones by the decimal length which immediately follows the letter S (for „set“). This way, it is clear that there is the set origin after the comma, and then the X (hexadecimal) set representation. If there is no length after the S (that is, if the comma follows immediately), we have the old syntax.

So it is possible for the P-Code interpreters or -compilers to handle both the old and the new syntax.

The next task deals with the two instructions SLD and SMV (set load and set move).

Here we have the old description of these instructions (from the 2019 P-Code paper):

SLD = set load

This instruction has two parameters: it loads a set constant or variable to the stack for further processing. p (the first parameter) is the length of the set in bytes, and q is the target address. The binary representation of the set (the source) is found at the address which is on top of the stack; this address was placed there before. The SLD instruction pops this address, moves p bytes from the source address to q, and pushes p and q.

SMV = set move

This instruction has two parameters: it moves a set from the stack to a set variable in storage. Both parameters (p and q) are length parameters. If the second length is lower than the first length, only the number of bytes specified by the first length is moved. The rest of the bytes of the target set variable is set to zero.

The sign of p may be negative or positive.

If p is positive,

- before the execution of SMV, the set (address and length) is on top of the stack; the target address is at the second position.

If p is negative,

- the absolute value of p is used as first length,

- before the execution of SMV, the target address is on top of the stack and the set (address and length) is at the second position.

SMV pops all two stack items in both cases.

An example:

```
LOC 46
LDA 1,424                       ; Y
LCA S,X20'00000000000000000000000000008000000200'
SLD 20,504
SMV 28,20
```

This is old syntax, of course.

In this example, you can see a constant set of length 20. Its address is first brought to the stack by LCA, then a SLD instruction is needed to build the old set representation on the stack (which consists of set address AND set length) and then this set is moved to another places; the address of the target has already been placed on the stack by LDA.

Using the new set representation, we can do much better, IMO:

```
LOC 46
LDA 1,440                       ; Y
LCA S4,120,X'80000002'
SMN 20,96
```

SMN means „set move new“.

There is no need for SLD any more in this case, because we have no special representations of sets on the stack; the set is simply identified by its address, and as long as the set is read/only, it can remain in the constant area.

Here the set is copied out of the constant area, and this is done using the new SMN instruction. It uses the addresses on the stack and it has two parameters, both rely to the target. The first parameter is the length of the target set (including the four bytes of meta information at the beginning) and the second byte is the set origin of the target set.

Both operands of SMN can be derived directly from the type of the target set. The implementation of SMN has to deal with different set origins (which means that the set representation of the source set has to be shifted correctly) and it has to check that all elements of the source set fit into the target set. More on this later …

Now: to support my idea of a smooth migration, I would like to create an intermediate compiler which does the following:

- create new set constants with origins etc. as outlined above

- but still create the old SLD and SMV instructions, with the lengths corresponding to the old set types (which don‘t have set origins, but instead all set elements stored starting with the element zero)

- then the implementation of the SLD instruction should be changed such, that it translates the LCA set constants of the new format to the old representation

But:

it just comes to my mind that this will not work, because SLD does not know if it has to load a set constant (which has the new format) or a set variable (which has the old format).

Let‘s see, if I can clone the SLD instruction into two different instructions; one for constants and one for variables … and then only the constant instruction must be changed to handle the new set constant format.

My goal is to create an intermediate compiler which incorporates some of the changes I made and still works :-) the best test of the compiler is to see if it compiles itself and produces the same code as its predecessor.

Some days later:

I deciced not to go this path any further, because I already created the new set format for the DFC constants, and they are moved to set variables (using block move instructions), so it is not possible any more to have a different format for set constants and set variables. BTW: I discovered this, AFTER establishing a new P-Code instruction SLC (= set load constant), which does the same as SLD, but for set constants instead of set variables. This is all moot, because IMO both instructions will not be needed any more in the near future.

Next steps:

- I‘ll generate the new instruction SMN instead of SMV, using the type attributes of the target
- The SLD and SLC instructions will be kept, but will be eliminated in the near future
- The problem is with the calls to FORCETEMPSET; this copies the set (constant or variable) to a temporary buffer and puts it address on the stack (using SLD/SLC); this is done today every time when a set is checked, but in the future, it is needed only during set manipulations (using + and * operations), but not – for example – for a simple IN check.
- Set moves between variables of the same set type could be done with simple block moves.

- Fix a simple error: the length of the set constants in the DFC instructions includes the 4 byte meta information, but the length in LCA S does not. It is better to NOT include the length of the meta information; so the length information only holds the length of the REAL set bit list.

List of existing set-related P-Code instructions

- ASE = add element to set
- ASR = add range to set
- CHK = check stack item (must be verified)
- DFC = define set constant (already modified …)
- DIF = difference of two sets
- EQU = equal comparison (set variant … same for other comparisons)
- IND = indirect access (set variant must be verified)
- INN = check if element is in set
- INT = intersection of two sets
- LOD = load (set variant must be verified)
- RET = return (can it really return sets?)
- SCL = set clear
- SLC = set load constant (new, see above)
- SLD = set load
- SMV = set move
- UNI = union of two sets

As mentioned above, the existing set instructions shall remain unchanged during the migration process, so that the existing compiler still works, using the old P-Code instructions. I already mentioned a new P-Code instruction called SMN (set move new) above, but now I am thinking of renaming it and putting all the new instructions in a separate „name space“ by giving them a new starting letter. Z comes to mind; we don‘t have instructions starting with the letter Z at the moment. So I would like to consolidate all new set instructions as instructions with the letter Z.

But there are some exceptions. First of all, I have to check, if some of the instructions above really work with sets (for example IND, LOD and RET etc.).
Second, LCA, DFC and the compare instructions are already known to work with sets as well as with other types and should work with old and new set representations simultaneously.

New set-related P-Code instructions

At the moment, we plan the following new set instructions:

- ZAE = add element to set
- ZAR = add range to set
- ZDI = difference of two sets
- ZIN = check if element is in set
- ZIS = intersection of two sets
- ZCL = set clear
- ZMV = set move
- ZUN = union of two sets

I don‘t know yet, if all of these instructions are really needed. And if some instructions are still missing.

The next steps:

- rename SMN to ZMV
- create ZIN instead of INN
- omit the SLC and SLD instructions
- write a new PCINT version, which implements the instructions so far, and look if it handles the test programs TESTSET9.PAS and TESTSETA.PAS correctly (this will sure take some days or weeks)

Builtin functions

When working on the new P-Code instructions. I discovered some other areas which need rework:

- the CARD function has a set argument
- see procedure GENSETOP; it generates the set operations (new P-Codes) ZIS, ZDI and ZUN … but it also does other work on the stack representations, which maybe is not needed any more or needs changes …

I thought about telling somewhere in the P-Code if the compiler uses the old or the new set implementation … this way the interpreters and P-Code translators would know how instructions that stay the same (like the comparison instructions etc.) should be handled.

A flag which tells this could be included in the function prolog, that is: in the P-Code instruction ENT, if there is space remaining. Otherwise, another new mode instruction, which shows several flags for the compile unit, could be established.

This flag could also control the behaviour of the builtin functions mentioned above.

02.09.2024

In the meantime, I changed the compiler to output the ZMV and ZIN instructions (and DFC and LCA for sets, of course), and I changed PCINT to understand them. Now the test programs TESTSET9 and TESTSETA work correctly, even for sets involving negative integers. But some problems still remain, because sets with base type CHAR are not handled correctly. I am trying to solve this in the next days.

Every set involving characters should only create constants using the C notation (because of portability concerns); this is also true if the base type is not CHAR, but only a subrange of CHAR. This was not correct, as I observed this morning, but it could easily be fixed by introducing a new function IS_CHAR_TYPE (of type BOOLEAN), which returns true not only for CHAR types, but also for subranges of CHAR. This function must be used whenever sets have to be examined for a base type involving CHARs.

07.09.2024

I needed an additional new P-Code instruction. ZMX is much the same as ZMV, but it leaves the address of the result on the stack for further processing. This is needed prior to the new instructions ZUN (set union), ZIS (set intersection) and ZDI (set difference), because in these situations one of the sets involved in the operation has to be copied to a work area first, and this is done using ZMX.

Today I finally have a working version of the compiler which produces correct code for ZUN, ZIS and ZDI operations … and they also work and give correct results with PCINT. This is a major breakthrough and shows that my design is valid – so far.

When trying to compile the compiler, I get some problems with such statements:

```
ERRLOG := ERRLOG + [ FERRNR ] ;
```

where ERRLOG is a set with base type INTEGER and FERRNR is an integer variable. At the moment, the new compiler abends, when compiling this … the old compiler compiled this without problems.

This must be fixed in the next step … the compiler tries to check the properties of the right side operand of the set union, but obviously it can‘t do it, because the value of FERRNR is not known, and so there are no properties (set origin and length etc.). The old compiler had no need for set properties … all sets started with element zero, so it could implement this without requiring information about the set properties (maybe the element was outside the set … but who cares).

The solution to this is the (former) P-Code instruction ASE, but this instruction must get a new equivalent, as the other old set instructions, which are all obsolete now.

Same goes for expressions with a mixture of constants and variables; let‘s assume, the expression above would loke like this:

```
ERRLOG := ERRLOG + [ 12, FERRNR, 15 ] ;
```

or like this:

```
ERRLOG := ERRLOG + [ FERRNR1 .. FERRNR2 ] ;
```

More to do in the next weeks :-)

The rework of the set implementation is finished, when the compiler is able to compile itself and it is successfully running (that is, it can repeatedly compile itself and produce identical results – and the compiler uses Pascal sets heavily; for example: the symbols which may restart parsing after a syntax error in a certain situation are stored in a Pascal set).

I hope you like the new set implementation of New Stanford Pascal;
please send comments and suggestions to

berndoppolzer@yahoo.com

or

bernd.oppolzer@t-online.de