Paper Discussed: Dependent Types for Low-Level Programming (Deputy).

In previous papers that we have talked about the soft bounds, baggy bounds methods in which the compiler inserts the code to check the out of bound errors. So in the Deputy paper we talk about the method to check the bounds without adding anything by the compiler and by checking with all the code given by then programmer.

**Dependent Types:**

**Key idea:** Verify the bounds of checking code that programmer already wrote.

Rather than adding any new code, this method deal will check the bounds with the already existing code.

For example:

Program 1:

Suppose the programmer wrote:

```c
void mycopy(int *dest, int *src, int n)
{
    int i;
    for(i=0;i<n;i++)
    dest[i] = src[i];
}
```

There is some info about the bound in the parameters, which is in n. So the bounds information is already present in the parameters.

```c
void mycopy(int *COUNT(n) dest, int *COUNT(n) src, int n)
{
    int i;
    for(i=0;i<n;i++)
    dest[i] = src[i];
}
```

This means that dest is a pointer to n items and src is a pointer to n items. This is a requirement and not a promise.

**Program 2:**

```c
void foo(void)
{
    int A[11],B[7];
    mycopy(A,B,6);  ----(1)
}
```
Statement (1) is allowed. Compiler in order to guarantee these annotation are safe, it has to understand these annotations. It has to check that mycopy(A,B,6) in program 2 meet the requirements in program 1(\texttt{void mycopy(int \*COUNT(n) dest, int \*COUNT(n) src, int n)}). This is done by deputy.

The way the deputy uses is:
In programming terminology we write
Dest: ptr(n) int
Dest is a pointer to n int’s.
The big deal with dependent types is that when we have, int or union then the problem with this type is that they do not mention other variables of the program.

\textbf{For example:}
A: ptr(11) int // this does not mention other variable in the program i.e. n.

The compiler has to make sure that mycopy(A,B,6) matches with \texttt{void mycopy(int \*COUNT(n) dest, int \*COUNT(n) src, int n)}.

The compiler wants to know whether A is a pointer to 6 ints. This is the question that compiler has to answer.
A: ptr(6) int -----(2)

So what do we knw abt A. We know that A is a pointer to 11 integers.
A: ptr(11) int -----(3)

Now the compiler wants to know that that of A is a pointer to 6 integers then is it a pointer to 11 integers. In other words does command (3) imply command (2). The compiler wants to know whether it point to 6 as well.

\begin{verbatim}
void foo(void)
{
    int A[11],B[7];
    mycopy(A,B,C);
    A: ptr(11) int
    A: ptr(6) int
}
\end{verbatim}

\textbf{General Rule:}
The paper talks that if the thing above the line is true then the thing, which is below the line, is also true.

\textbf{General Rules of the paper:}
P : ptr(n) int n>=m
P: ptr(m) int
So according to general rule, from the above example we have to check if 11 >= 6.

**Programming by Contract:**
This means that we have to specify the requirements of a module. We have to check if the requirements are met then the module will function correctly and then we will check whether we call that module or not. When we call the module we need the requirements. So in the above example we have to check that whether mycopy(A,B,C) meets the requirements.

**Evaluation Criteria:**

1. **Effectiveness:** It is going to check for the bugs when ever we call a function like void mycopy(int *COUNT(n) dest, int *COUNT(n) src, int n). There will be no spatial bugs. If every function is verified with what it is provided and not adding any extra stuff then there will not be spatial bugs. But we do have temporary bugs. So this is effective.

2. **Portability:**
3. **False Positive:**
4. **Separate Compilation:** Good in separate compilation.
5. **Binary compatibility:** It means that it may insert some checks in the body of the function at the runtime but deputy never modifies the interface. Deputy does not introduce any extra bounds checking like in soft bounds. That is a compatibility problem. The deputy does not add the bounce table. It just wrote that what the programmer already wrote. It is not going to use any new metadata. All the checks will happen with all the data that is already present i.e. with the entire variable that is already present. So very good compatibility.

6. **No code changes:** Here the programmer has to do lot of code changes. But this is a better option compared to previous i.e. better than baggy bounds. As here same same binary is generated. So the advantage of this is that we do not break the code. The problem with baggy bounds is that there is a probability that the code gets broken.

7. **Performance:** Imagine that the programmer has written ‘COUNT’ thing and every time the programmer makes the call the compiler has to verify that every dereferencing is safe so it does not has to insert any extra code to increase the performance. So in this case the run time overhead is zero, as we would be having the same binary. So no baggy bounds are required or no extra code is required. So overhead is quite low. In the paper the overhead shown is 6%.

In Linux kernel there are two kinds of pointer. First one is the kernel produce a pointer like kmalloc, which points in memory. Second is the pointer that comes from user land like the pointer used in read, write and we pass pointer to a buffer. The buffer lives in memory space, so we can’t pass that into kernel as that will cause the problem as we can’t play with read, write in kernel memory. There are tools that detect these kinds of bugs and require the change of code like struct can be used in both kernel and user.

If there is a struct the programmer has to make two copy of struct like,

1. `struct_foo_userversion`
Verification that the call is safe

Now we have to verify that

```c
for(I = 0;i<n;i++)
{
    dest[i] = src[i]
}
```

- It is safe

<table>
<thead>
<tr>
<th>Src:ptr(n) int</th>
<th>i:int</th>
<th>0&lt;= i&lt;n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src[0]:int</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

This ensures good compatibility. But we can do better.

New Concept:
Loops are complicated. We can analyze from ‘for loop’. The simpler version of loop for e.g. is(there is a new concept of weekest point ……)

The compiler will keep track of run time state of program

**Example 1:**

<table>
<thead>
<tr>
<th>Fact</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x=3}</td>
<td></td>
</tr>
<tr>
<td>{x=5}</td>
<td>x=5</td>
</tr>
</tbody>
</table>

If suppose the compiler says x= 5; the fact will say {x = 5};
If suppose that there is x= 3 previously then it has to erase that fact to change the value to 5;
Example 2:

<table>
<thead>
<tr>
<th>If(pred)</th>
<th>The compiler knows the fact that pred is true in if.</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>The compiler knows the fact that pred is false in the else.</td>
</tr>
</tbody>
</table>

We can write these few fact to know the run time state of the program.

So the fact in this if we have an expression like:

Example 3:

<table>
<thead>
<tr>
<th>for(i=a; i&lt;b; i+=c)</th>
<th>The fact here is {a&lt;=i&lt;b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

The assumption here is that the compiler has to confirm that the value of i is not changed in the loop. Because if the value of i changes then it is not valid. Suppose in the program the programmer does:

Example 4:

```c
for(i=a; i<b; i+=c)
{
    p=&i;
    -
    *p+=7;
}
```

In the above example p is pointing to i and if the value of p changes then it is modifying the value of i. This is called aliasing problem. It is hard to determine whether the variable i has changed the value or not. This loop is modifying the value of i. This is difficult for the compiler to know. There are some rules for the compiler so that i is not modified in the loop.

First idea: we can make a copy. But this is a run time check.
Second idea: if i is a local variable, if the program never writes &i, and &i never taken and i not modified in loop. These two steps when considered then we can make sure that the value of i never change. This is compilation time check

**Finally we have:**

```c
for(i = 0;i<n;i++)  // fact will be {0<=i<n}
{
    int i;
    for(i=0;i<n;i++)
        assert(0<=i && i<n)   // no need of this
    dest[i] = src[i];
}
```

And we have to get rid of the run time check, which is asset for this we will use the concept discussed above. The compiler will ask, does this {0<=i<n} fact imply the assertion will always succeed. If the compiler can make sure that this then we do not need the assertion. This is a hybrid type system.

The key idea is to use the bounds checking code that the programmer already wrote. There are two methods in this:

1. We use the bounds information that the programmer has already written.
2. If checking for bounds then we would be using the programmer’s bounds checking code to prove that assert is true.

**Example:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{</td>
<td>The compiler will have the same fact</td>
</tr>
<tr>
<td>int i,j = 10;</td>
<td>{0&lt;=i&lt;n}</td>
</tr>
<tr>
<td>for(i=0;i&lt;n;i++)</td>
<td>Here the compiler has a new fact</td>
</tr>
<tr>
<td>assert(0&lt;=j&amp;&amp;j&lt;n);</td>
<td>{0&lt;=i&lt;n}</td>
</tr>
<tr>
<td>j=i;</td>
<td>&amp;&amp; i=j</td>
</tr>
<tr>
<td>dest[i] = src[i];</td>
<td>The compiler will look at the facts 0&lt;=i</td>
</tr>
<tr>
<td>}</td>
<td>i&lt;n</td>
</tr>
<tr>
<td>Src:ptr(n) int j:int 0&lt;=j&lt;n</td>
<td>i = j</td>
</tr>
<tr>
<td>Src[i] : int</td>
<td>The compiler will ask whether these facts imply that 0&lt;=j &amp;&amp; j&lt;n. If yes which is the case now then it there is no need of assert.</td>
</tr>
</tbody>
</table>
**Example:**
Suppose we have a function:

```c
Int myprint(int *COUNT(n), int n)
{
    while(n>0){
        printf("%d", *p); // no printf bugs here
        p++;
        n--;
    }
}
```

**Problem with this code:**

p: ptr(n) int
p is the pointer to n integers;

So is p still the pointer to n integers after its increment? If p is a point to integers it is always a pointer to int. After incrementing the value of p, it is not the pointer to n integers but a pointer to (n-1) integers.

Here, n++ is also not allowed.

The deputy has figured a way to deal with this.

What we can do is:

There are two things going on here:

Whenever we update the values we will update them simultaneously or we will perform the assignment simultaneously. **This is called a parallel assignment.**

We verify that the new types meet the old types i.e.

P: ptr(n) int  n: int  n>0

(p+1):ptr(n-1) int

The new value of p. p must be pointer to n integers. The new value of p is p+1 and that of n is n-1. The new value of p i.e. (p+1) is a pointer to (n-1) integers.

**Advantage:**

This method does not modify the program.