Conventional Scoping of Registers – An Experiment in ε_χTeX

Gerd Neugebauer

Abstract

TeX provides groups as a means to restrict the visibility of registers. This construction is well known in the TeX world but does not coincide with the groups as known from other programming languages. If we refain from string the register value in a global array we can come to the alternate solution of storing it in the control sequence used to access it. With this variant we can provide a meas to define an arbitrary number of registers which follow the same scoping rules as the variables in Pascal-like languges.

ε_χTeX is a reimplementation of TeX in Java. It is developed with the extensibility and confgurability in mind. The idea of an alternative storage for registers can be implemented in ε_χTeX as an extension. It is shown which steps are required for such an implementation. In this course the extensability of ε_χTeX is demonstrated.

1 Storage in TeX

TeX stores the values of registers in TeX memory.

2 Registers and Scoping

`plain.tex` prvides macros to handle the allocation of registers. For this document we want to restrict our considerations count registers. Here the macro `\newcount` can be used to allocate a new count register:

```
\newcount\abc
{\abc = 42
 \showthe\abc
}
```

The

```
{ int abc = 42;
  printf("abc␣=␣%d", abc);
}
```

3 ε_χTeX

4 Writing a New Primitive for ε_χTeX

According to our considerations we want to have a new primitive which behaves like a count register but stores the value within the code and not in the context. In addition we need a primitive `\integer` to dynamically create new such integers. Then we can write the following TeX code:

```
{\integer \abc = 42
 \showthe\abc
}
```

First we start with implementing the code for the count equivalent. This code needs to have several properties to behave like a count register:

- It needs to assign a new value when executed. This means that

  `\abc=123`

  works if `\abc` has the meaning of the new primitive.
- It needs to act as an assignment; this means that `\afterassignment` as to be taken into account. This mean its token is expanded after the assignment has taken place.
- It needs to be advancable. This means that the following works:

  `\advance\abc by 123`
- It needs to be multiplyable. This means that the following works:

  `\multiply\abc by 123`
- It needs to be dividable. This means that the following works:

  `\divide\abc by 123`
- It needs to provide the count value upon request. This means that the following works:

  `\count0=\abc`
- It needs to provide value for primitives `\the` and `\showthe`. This means that the following works:

  `\showte\abc`
- It needs to expand to the tokens making up its value.

5 ε_χTeX

The ε_χTeX project (→ http://www.extex.org) has the aim to provide a reimplementation of TeX. The implementation language for this reimplementation is Java. The major design decisions put the modularity and configurability into the center.

6 Providing a Definition

To start with we create a new class. This class lives in a package named `extex.tutorial`. In addition we use a bunch of imports from ε_χTeX. Since the imports are usually filled in by the IDE, we omit them (like the comments which are assumed to be filled in by the reader).[1]

```
package extex.tutorial;

import org.extex.core.count.Count;
// a bunch of more imports omitted
```

[1] To be honest, the exact package structure of ε_χTeX is subject to some changes until the final version 1.0 is released.

Next we declare the class. It is derived from an abstract base class which takes care of the assignment. Each of the properties we want to have is declared with the help of an interface. `Advancable` describes that the primitive can be used after the primitive `\advance`, `Dividable` describes that the primitive can be used after the primitive `\divide` and so on. Each of these interfaces contains a single method which needs to be implemented.

```
public class IntPrimitive
        extends
            AbstractAssignment
        implements
            Advanceable,
            Divideable,
            Multiplyable,
            CountConvertible,
            Theable,
            ExpandableCode {
```

Since we want to store a count value with the code we first create a private field. The data type `Count` encapsulates a count value. It has the methods to access and manipulate it. In it's core it contains a `long` value to store a number in.

```
private Count value = new Count(0);
```

But before we come to implement the interfaces we have to define a constructor. The constructor takes one argument – the name of the primitive – and passes it to the constructor of the super-class.

```
public IntPrimitive(String name) {
    super(name);
}
```

Now we can start with the first method `assign`. It takes four parameters with the following classes:

`Flags` contains the indicators for the prefix arguments like `\global`. The primitive can consume the flags and react differently upon their values. Since out primitive does use prefixes this argument is simply ignored.

`Context` contains the equivaent to the TeX memory. Anything contributing to the state of the interpreter is stored in the `Context`. This `Context` is also stored in a format when `\dump` is invoked.

`TokenSource` provides access to the scanner and the parsing routines. It can be used to acquire further tokens or even higher order entities.

`Typesetter` contains the typesetter of the system. The typesetter produces nodes which might be stored in boxes and finally sent to the backend.

These parameters will come back for the other methods.

```
public void assign(Flags prefix,
          Context context,
          TokenSource source,
          Typesetter typesetter)
        throws InterpreterException {

    source.getOptionalEquals(context);
    Count newValue = CountParser.parse(
        context, source, typesetter);
    value.set(newValue);
}
```

The implementation first consumes an optional euqal sign and then parses a following count value. Finally we can set the internal count to this new value.

Assume that we have assigned the new primitive to the control sequence `\abc` – somethong which we will reveal later. Then we can do the following:

```
\abc = 1234
```

This assigns simply a new value to the variable. But we have also used the infrastructure of an assignemnt. Thus the token stored in the token register `\afterassignment` are inserted after the assignment:

```
\afterassignment=\x
\abc = 1234
\y
```

Right now we can assigne a new value to the variable. Since we want to see what we have done we implement the method `the` which converts the value back into tokens to be used by the primitives `\the` and `\showthe`.

```
public Tokens the(Context context,
        TokenSource source,
        Typesetter typesetter)
      throws InterpreterException,
             ConfigurationException {

    return value.toToks(context);
}
```

Next we have to take care of `\advance`. In ε_{χ}TeX the implementation of `\advance` decouples the operation from the implementation of the primitive. Tus it is posible to add further primitives which can be used after `\advance`. This goal is reached by providing the interface `Advancable`. When the token as the meaning of code which implements this interface then the control is passed to the methods defined in the interface to carry out the operation. We use this feature to make our primitive applicable for `\advance`.

The method used the parsing routines in ε_{χ}TeX to acquire the optional keyword `by` and the value for a count register. Tis value is added to the variable stored in this primitive.

```
public void advance(Flags prefix,
          Context context,
          TokenSource source,
          Typesetter typesetter)
        throws InterpreterException {

    source.getKeyword(context, "by");
    Count by = CountParser.parse(
          context,
          source,
          typesetter);
    value.add(by);
}
```

The same technique used for `\advance` is used for `\divide` as well. Thus we just have to implement the associated interface `Dividable` and provide the following method:

```
public void divide(Flags prefix,
        Context context,
        TokenSource source,
        Typesetter typesetter)
      throws InterpreterException {

    source.getKeyword(context, "by");
    Count by = CountParser.parse(
          context,
          source,
          typesetter);
    value.divide(by);
}
```

And once again the same trick for `\multiply`: We implement the interface `Multipliable` and provide the following method:

```
public void multiply(Flags prefix,
        Context context,
        TokenSource source,
        Typesetter typesetter)
      throws InterpreterException {

    source.getKeyword(context, "by");
    Count by = CountParser.parse(
          context,
          source,
          typesetter);
    value.multiply(by);
}
```

Converting into a count value is expressed with the interface `Countconvertible` which has one method `convertCount`. This method delivers the count value as `long`. Since we have the variable in our private field we can just take the value from there.

```
public long convertCount(
        Context context,
        TokenSource source,
        Typesetter typesetter)
      throws InterpreterException {

    return value.getValue();
}
```

```
public void expand(Flags prefix,
        Context context,
        TokenSource source,
        Typesetter typesetter)
      throws InterpreterException {

    source.push(value.toToks(context));
}
```

This is all we need to do to implement the new primitive.

```
}
```

7 Putting Things into Place

Now we are finished writing out new primitive as a Java class. But how can we make use of it? First of all we have to compile it with a Java compiler and put it into a jar – say `abc.jar`. ε_χTeX is installed in a directory. This installation diretory contains a subdirectory named `lib`. All jars contained in this directory are automatically considered when classes are loaded. Thus we put `abc.jar` into this directory.

Next we make use of a quick extension mechanism to try out our fine new primitive. Later we will use the configuration mechanism of ε_χTeX for this ourpose. But now we simply use the dynamic extension mechanisn which allows us to bind some Java code to a primitive. To do so we need to load the unit `jx`. Units in ε_χTeX are collections of primitives. For instance there is a unit `tex` containing the TeX primitives.

One of the primitives contained in ε_χTeX – i.e. in the unit `extex` – is the primitive `\ensureloaded`. It takes one argument in braces which is the name of a unit and loads this unit of has not been loaded into the interpreter before.

This primitive is used now to load the unit `jx`:

```
\ensureloaded{jx}
```

After the unit `jx` has been loaded we can make use of the primitive `\javadef` provided by this unit. This primitive is similar to te primitive `\def`. It takes a control sequence and a list of tokens enclosed in braces. The control sequence gets a new meaning. This meaning is determined by the Java class named in the tokes argument:

```
\javadef\abc{extex.tutorial.IntPrimitive}
```

Now we can use the primitive `\abc` as shown in the beginning.

8 Defining new Variables

The definition of each new variable with `\javadef` is a little bit clumsy. We had the plan to define any new variable with `\integer`. It takes a control sequence and the initial value. This can be accomplished with a small definition of the following kind:

```
\def\integer#1{%
   \javadef#1{extex.tutorial.IntPrimitive}%
   #1}
```

This approach works but it has the disadvantage that his macro does not interact properly with `\afterassignment`. The primitive `\javadef` is an assignement. Thus the afterassignment token would be inserted just after the definition but before the intial value has been read.

To overcome this problem and gain some more insight into the definition of primitives in ε_χTEX we implement this primitive in Java as well.

```
public class IntDef
        extends AbstractAssignment {

public void assign(Flags prefix,
          Context context,
          TokenSource source,
          Typesetter typesetter)
        throws InterpreterException {

    CodeToken cs =
        source.getControlSequence(
            context,
            typesetter);
    IntPrimitive code =
        new IntPrimitive(cs.toString());
    code.assign(Flags.NONE,
                context,
                source,
                typesetter);
    context.setCode(cs,
                    code,
                    prefix.clearGlobal());
}
```

9 Configuring ε_χTEX

10 Conclusion