Working Paper Series
ISSN 1170-487X

A Calculator for supporting derivation in constructive type-theory: PICTCalc

by Steve Reeves

Working Paper 94/8

June, 1994

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

A Calculator for supporting derivation in constructive type-theory: PICTCalc [1]

Steve Reeves
stever@waikato.ac.nz
Department of Computer Science
University of Waikato
Hamilton
New Zealand

Abstract: PICTCalc is an interactive program written in LPA Prolog which has encoded within it the rules of Martin-Löf's constructive type theory (CTT), a formal system based on the constructive or intuitionistic mathematics of Brouwer, Heyting and others. It allows us to specify and express any total, computable function, so from a computer science point of view we can write both specifications and programs, along with the derivations which lead from one to the other, in a single language. PICTCalc is a more recent version of MacPICT which is itself a reconstruction of PICT [Ham92] and is intended as a test-bed for providing formal support for work within CTT. It has been developed and improved over the last five years, during which time it has been used to support teaching in an M.Sc. course on CTT at QMW. Many of the developments and improvements were suggested by students since they used the system to work on substantial courseworks. As we shall see, PICTCalc can be used to assist in the development of derivations in CTT and so it is called a derivation assistant (DA). In this paper I show how PICTCalc can be used for supporting derivations in CTT and consider what current experience suggests for future improvement. The examples in this paper will not require any knowledge of CTT since the meaning will either be clear to anyone with experience of logic and programming notations in general or will be explained as necessary.

Keywords: proof assistant, derivation assistant, constructive type theory, program correctness, specifications

---

[1] This working paper is to be published in a very similar form in the International Journal on Mathematical Education in Science and Technology.

1. Introduction

1.1 Background

Martin-Löf's constructive type theory (CTT) is a formal system based on the constructive or intuitionistic mathematics of Brouwer, Heyting and others. It is intended to encompass all of mathematics but, for the purposes of this paper we need only note that it allows us to specify and express any total, computable function,. From a computer science point of view we can write both specifications and programs, along with the derivations which lead from one to the other, in a single language.

PICTCalc (a Calculator for developing Programs and Proofs in Constructive Type Theory - so it should really be CalPPICTT) can be used to assist in the development of derivations in CTT. The outcome of a derivation is a conclusion (a judgement) which can be read as saying either that a particular program meets a particular specification (has a particular type) and so is correct or equivalently, via the "propositions as types" principle, that a particular proof proves a particular proposition.

PICTCalc is a reconstruction, by the author, of PICT, which was written by Alan Hamilton of Stirling University and most recently described in [Ham92], and is intended to support the teaching of CTT.

The development of PICTCalc has been guided both by the experience of surveying and judging other systems which have been developed to support teaching in formal areas of the mathematics and computer science curricula, see [GRB93] for examples, and by listening to students over the last five years as they used various versions of PICTCalc.

The students used PICTCalc during an M.Sc. course called "Constructive Mathematics and Programming" which involved a substantial amount of coursework. First they had to work through examples like those given later in this paper to get to know the system. Later they had to construct their own derivations for problems ranging from simple logical ones up to the development of functions over lists, for example sorting.

1.2 The system

PICTCalc is what is usually called a proof assistant, though it should really be called a derivation assistant (DA). That is to say it is an interactive program which has encoded within it the rules of Martin-Löf's constructive type theory (CTT) together with housekeeping functions, which make sure you are using rules properly, and ways of storing and retrieving definitions, theorems and derivations so that large problems can be worked on in a modular and tractable fashion. It also, to a certain extent, can be made to do trivial things automatically, so relieving the user of tedious and repetitive work.

As far as a user is concerned PICTCalc consists of a set of tactics, which are the type-theory rules "used backwards" in a goal-oriented approach to constructing derivations; a set of navigation commands for moving around the derivation; a set of actions, which allow the user to do things like save and restore parts of a derivation and switch between interactive and

automatic modes; a set of information commands which allow the user to view parts of the derivation; and a help facility which describes each of the available commands. In addition to these PICTCalc specific features, the system also uses the standard Mac editing capabilities which make handling the often complicated expressions of CTT much easier than would otherwise be the case.

The system runs on all Macs, though the amount of RAM available will determine the size and complexity of any derivation constructed. It is available free of charge and a copy can be obtained by making a request to the author.

2. CTT in PICTCalc

2.1 Syntax

One problem that becomes evident once you start reading about CTT is the abundance of different syntaxes. I can think of at least five different syntaxes that have been used in presenting Martin-Löf's work, and I have no doubt there are others. This poses the implementor of a system like PICTCalc with the problem of which syntax to use. I have been guided partly by my own preference (of course) and partly by what was already available in PICT. Happily, I was guided to almost the same place.

The "published" syntax closest to PICTCalc's is that used in [Bee85] where there is a clear and useful discussion of it. This syntax is to be preferred because, to paraphrase Beeson, it moves technical worries about bound and free variables into the derivation assistant and out of the object language, which is how it should be.

As long as the reader has experience of at least one of the accounts of CTT there will, in any case, be no problem interpreting the syntax used in PICTCalc.

Examples, following an obvious pattern, are:

N, 0, succ, rec

Falsity, case

Truth, unit

Bool, true, false, if

∏, lambda, ap

->, lambda, ap

∑, (a,b), split

/\, (a,b), split

\/, inl, inr, when

I, eq

List, nil, cons, listrec

U

rec(0,0,(x1,x2)@succ(x1))

[Note here the use of '@' to explicitly denote abstraction, as in PICT]

x, y, z, a, b, c, x1, x2 etc. as variables

Some symbols do not appear on the usual keyboard, and they have the following key combinations:

∏ needs shift-option-p

∑ needs option -w

/\ needs slash followed by backslash

\/ needs backslash followed by slash.

2.2 Judgements

A judgement can have one of the four following forms:

A type H

A = B H

(a = b) : A H

a : A H

with the usual meanings in each case and where the list H contains assumptions. Note the parentheses around "a = b" and the use of ":" to denote elementhood.

As usual, judgements where H is non-empty are called hypothetical judgements and are denoted by expressions like

rec(x, y, (x1,x2)@y) : I(N,x,0) [x : N, y : I(N, x, 0)]

where, as in CTT, the assumptions (in square brackets) are ordered so that a variable appears exactly once on the left-hand side of some assumption *before* it appears anywhere else in the assumption list.

2.3 Rules

Many rules have an extra premise which says something about the well-typedness of a term. For instance the CTT rule N-elim is

$$\frac{c : N \quad d : C(0) \quad e(x,y) : C(succ(x)) \ [x : N, y : C(x)]}{rec(c,d,e) : C(c)}$$

and note here that nothing is said about what C is. This is because, following Martin-Löf's original style e.g. [M-L84], the rules are abbreviated in many cases. This abbreviation amounts to showing only those premises of a rule which, as judgements, are of the same form as the conclusion. Also, in CTT only those assumptions which are discharged on use of the rule are shown. (There is the special case of the assumption rule where the only assumption shown is the one introduced by use of the rule.)

In our particular example, the premise "C(x) type [x : N]" is omitted. In PICTCalc, however, the rules are never abbreviated and so N-elim will have four premises.

3. Running the system

3.1 Introduction

After starting up the system your screen should look something like figure 1.

The `File` and `Edit` menus hold no surprises. The others will be treated in subsequent sections.

Eventually the `current node` window will, during the building of a derivation, display the contents of the node with reference to which any command you choose will be carried out.

Typically this window will have the form show in figure 2.

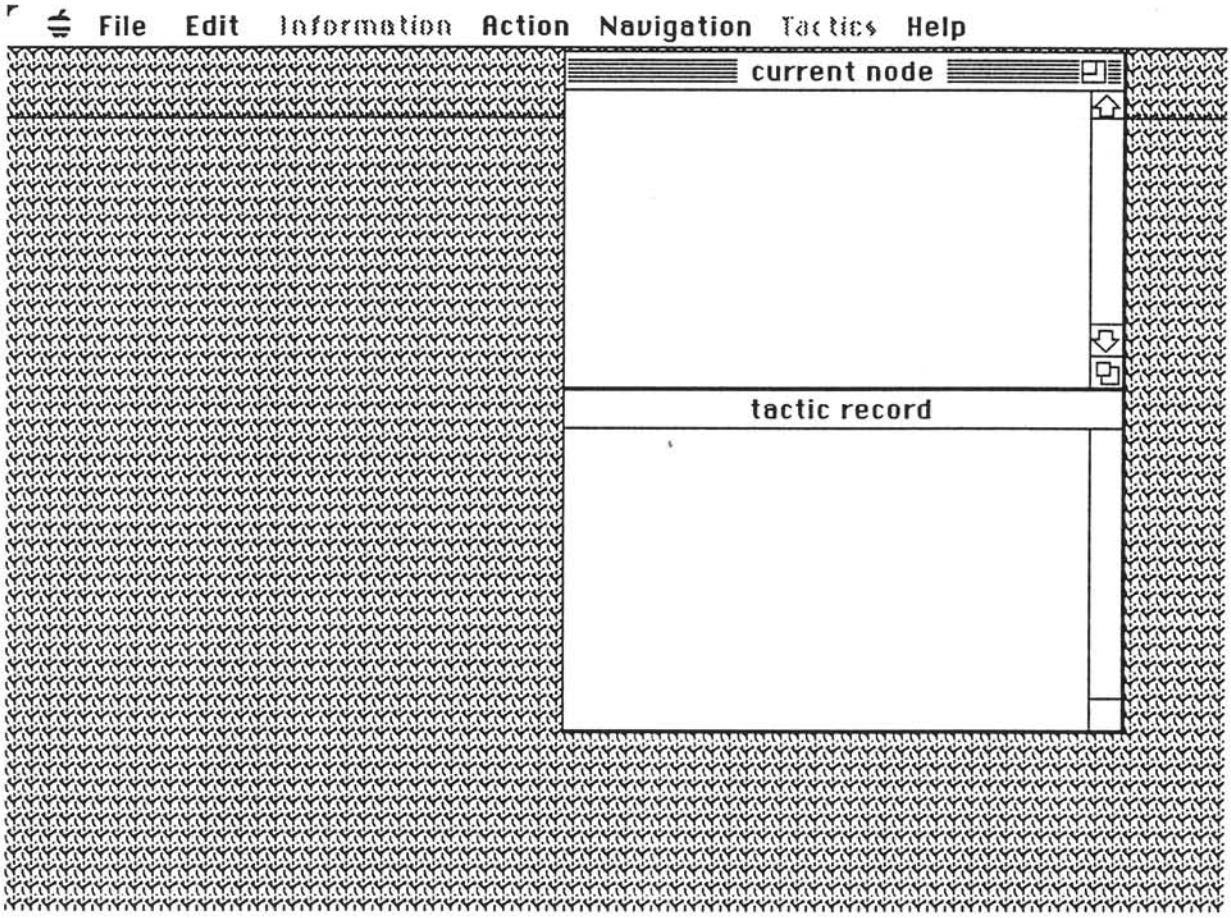


Figure 1

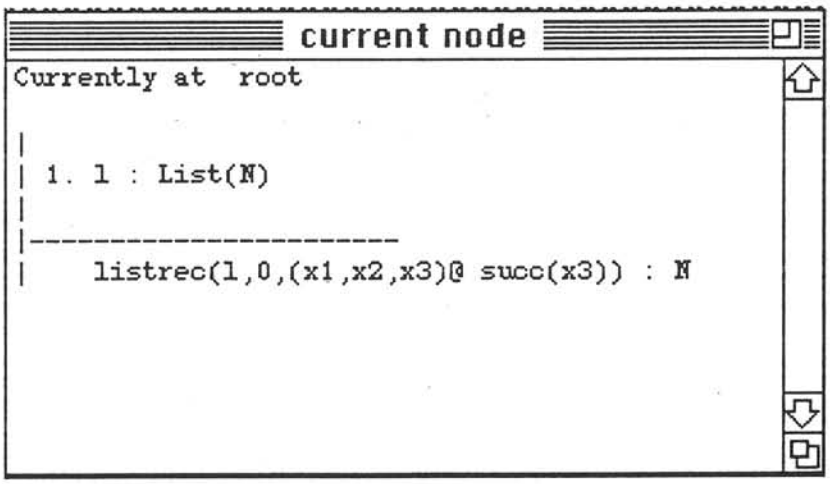
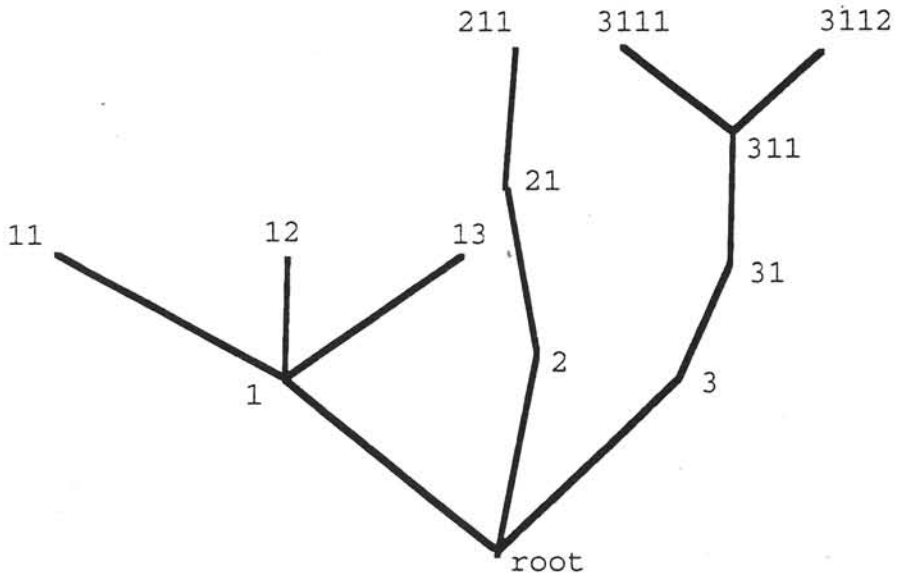


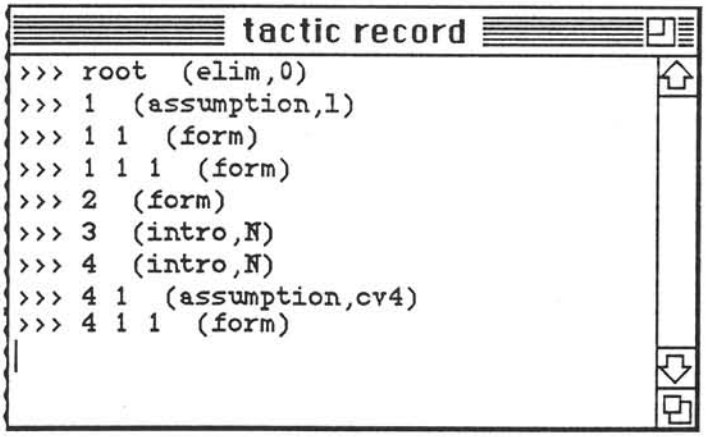
Figure 2

The first line gives an address for the node. In general an address is a list of integers which describes the position of the node in a derivation as shown in the following diagram:



Next, the lines labelled 1., 2. are the assumptions for this node. They are followed by a horizontal line and then, finally, by the conclusion for this node.

The tactic record window records the names of tactics used at each node in the current derivation as follows:



Here each line consists of an address and the name of a tactic along, perhaps, with other information. So, at node 1 1 in the derivation above the tactic `form` was used and at node 1 the `assumption` tactic was used to introduce the variable `1`.

All of the windows that appear in PICTCalc can have their contents copied, so that once the problem has been set up further typing is reduced to a bare minimum. You should find that almost anything that you have to type already appears, or can be made to appear, somewhere in some window.

You should note that at this stage in the execution of MacPICT there are only a few options that you can choose from. Many options are simply not appropriate and this is denoted (and you are disallowed from using them) by showing them in grey on the menu bar (if whole menus are disallowed) or in their respective menus.

A description of each menu and the options within it are given in the appendix. In the body of the paper I will be giving examples which show how some of these options are used in practice.

4. Definitions, theorems and derivations

4.1 Definitions

One important way of making a derivation easier to work with is by sensible use of definitions, which are pairs of the form (definiendum, definiens).

As well as simply making expressions textually smaller, which enhances readability and hence comprehension, they also, by choice of suitably suggestive definienda, make expressions more comprehensible.

Within PICTCalc we can associate a definiendum with a definiens by using `newdef` from the `Action` menu. This adds the definition to the current internal database of PICTCalc. A definition can be removed from the current database by using `rmdef`.

The current definition database can be saved in a file on disc by using `savedefs` and previously saved definitions can be loaded into the definition database by using `loaddefs` from `Action`.

We can see any particular definiens by using `showdef` from the `Information` menu.

When we need to replace a definiendum by its definiens we can use `unfold1` from `Tactics` which extends the current derivation by creating a new goal which is the same as the current excepting that all occurrences of the definiendum chosen when using `unfold1` are replaced by the corresponding definiens. The option `unfold` simply has the effect of applying `unfold1` to all definienda that appear in the current goal.

4.2 Theorems

Another important way of working with derivations by hiding detail and structuring them in a natural way is by using theorems.

These are simply previously derived judgements which are stored in an internal database and can be used to extend the current derivation. Since, in general, the judgement may be hypothetical, any assumptions appearing in the judgement are used as sub-goals to extend the derivation.

As with definitions, there are commands for making a new theorem (out of a completed derivation), looking at theorems stored in the database, loading a file of theorems into the database, saving the contents of the current database to disc and removing theorems from the database.

The relevant commands are `thm`, `newthm`, `showthm`, `loadthms`, `savethms` and `rmthm`.

4.3 Derivations

Often it turns out that you need to save a derivation. This may be because you have run out of time or because (as when editing) you are nervous about the robustness of your system and want to make periodic backups of your work so far.

Also, you may reach a stage in a derivation when you realise that another theorem would be useful, which means you want to stop the current derivation, do another and save its result as a theorem and then resume the first derivation, perhaps using the theorem just derived.

In order to allow this way of working the options `savederivation` and `loadderivation` are provided in the menu `Action`.

Eventually it is intended that the information stored by `savederivation` will be used to reconstruct the derivation in a 2-dimensional form.

5. Some examples

In order to introduce the use of PICTCalc, we will look at some simple examples. The first set of examples will be "logical" in the sense that they will involve types that have a clear logical interpretation. The second set will be more clearly about computational objects and their types.

Note that the particular occurrences of expressions of the form "cvn" or "vk" in the examples below may differ from the occurrences you see (in the value of n or k) since they depend on previous activity with PICTCalc.

5.1 Logic

The first example shows how we can build a derivation for A -> (B -> A). In the course of this derivation we will construct an object in A -> (B -> A), i.e. a proof object, which is what it means to show that A -> (B -> A) is provable in CCT and so intuitionistically or constructively true.

First, choose start from the `Action` menu (shortcut ⌘S). Since we want to prove a proposition, i.e. construct an object in it, we choose `construct` from the first dialogue that now appears and since we want to input the type from the keyboard we choose `user` from the second dialogue.

Then, we are prompted for the assumption list. Since A and B in the proposition are intended to be any propositions, i.e. any types, we want to say that they are simply objects from the universe U, so the assumption list is

[A : U,B : U]

Having pressed OK for the assumption list we then type

A -> (B -> A)

for the type that is asked for next.

After some time the screen will look as shown in figure 4.

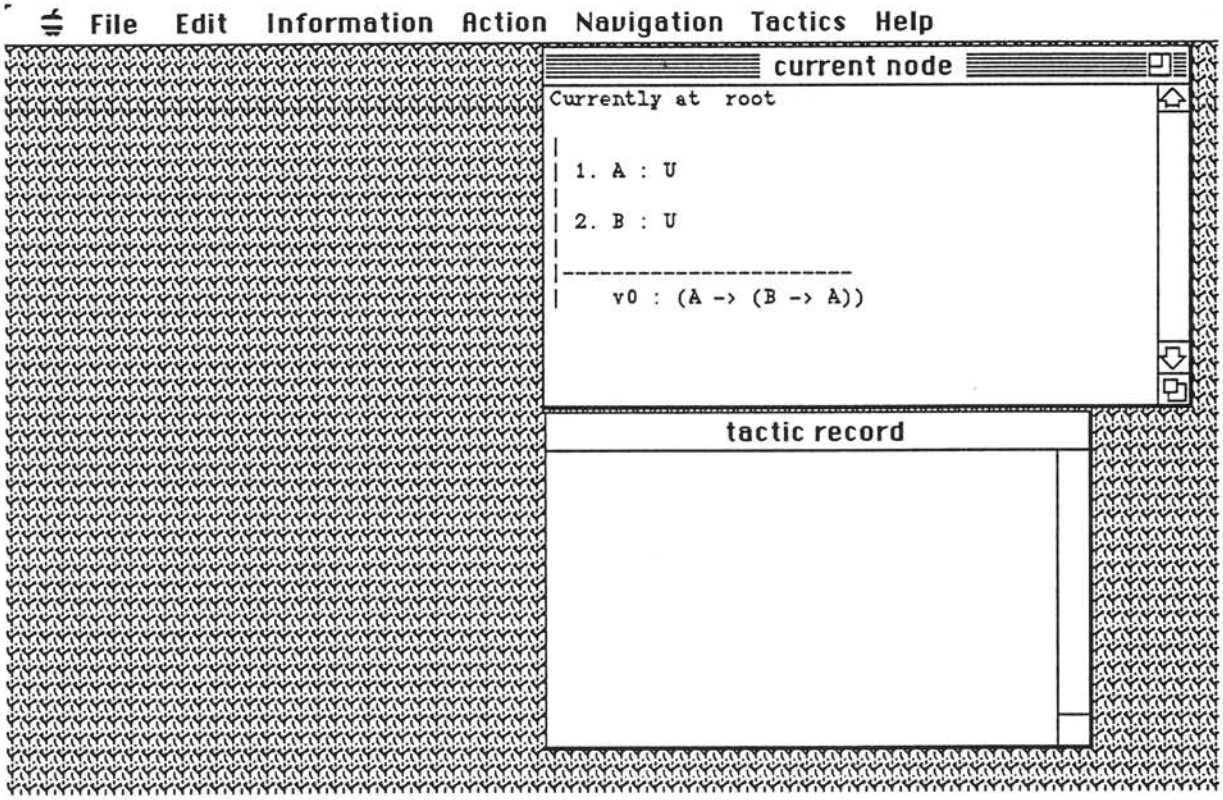


Figure 4

The v0 is an example of a scheme variable. This is a syntactic object invented by PICTCalc and it can stand for any CTT object. It is not itself from the language of CTT.

This first goal, then, says that something as yet to be built, whose place is currently held by v0, is an element of A -> (B -> A).

We now look to see what rules have a conclusion that matches this conclusion, remembering that v0 can stand for any object. An obvious rule that does match is ->-intro, so we can go to the `Tactics` menu and choose `intro`. This succeeds and changes the display, as shown in figure 5.

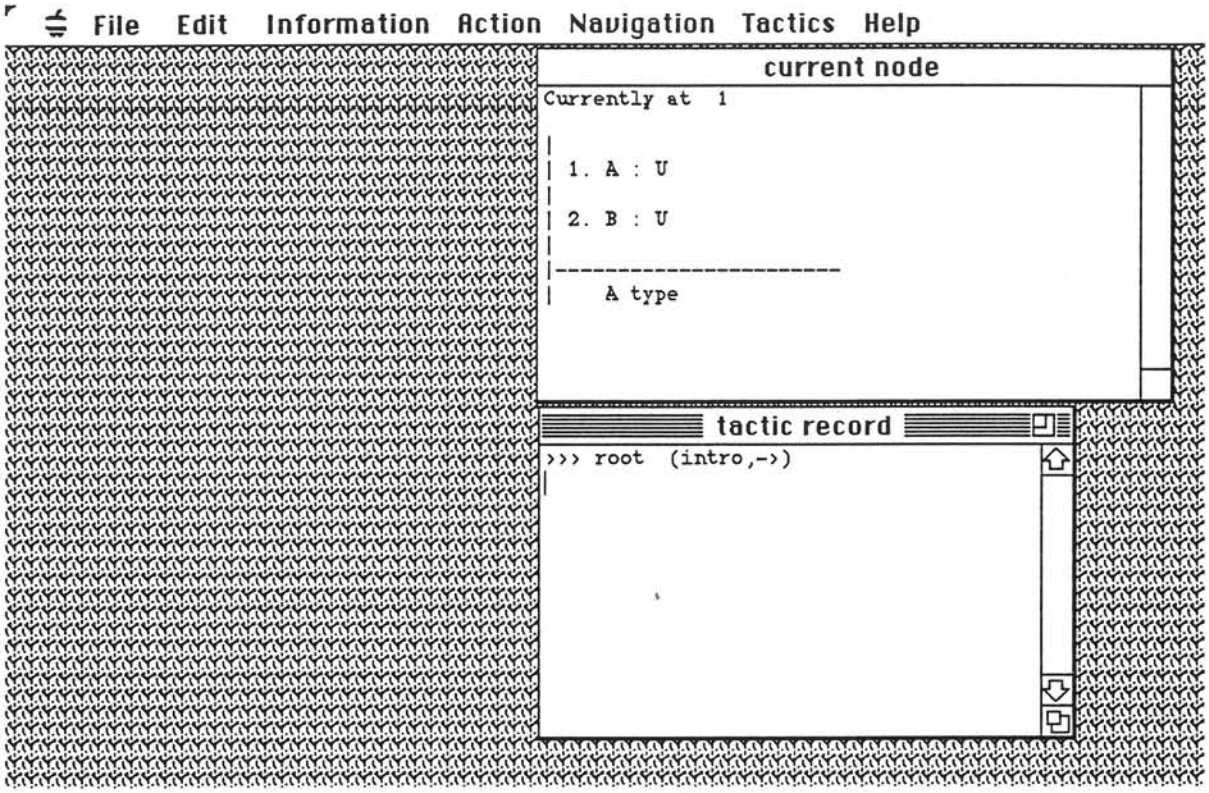


Figure 5

To see why this has happened we need to look at the rule ->-intro:

$$\frac{C\ \text{type} \qquad D\ \text{type} \qquad d(x) : D\ [x : C]}{\text{lambda}(x@d) : C \to D}$$

The matching that has gone on here is that v0 matched with lambda(x@b), C matched with A, D matched with (B -> A) and a new scheme variable v1 matched with d(x), which gives us three sub-goals:

A type [A : U,B : U]

(which is the one shown in the figure above) and

(B -> A) type (1)

and

v1 : (B -> A) [A : U,B : U,cv3 : A] (2)

which we will see in a moment.

To satisfy the goal at 1 we need to choose `uelim` from `Tactics`, which applies the rule

$$\frac{A : U}{A \text{ type}}$$

giving the sub-goal

A : U [A : U, B : U]

which is in turn satisfied by `assumption` and then, with no further sub-goals, by `form`.

To get to the next original sub-goal, i.e. the one numbered (1) above, we choose `next` from `Navigation` (shortcut ⌘N). To satisfy this goal we need `form`, followed by `uelim`, `assumption`, `form`, `next`, `uelim`, `assumption`, `form`.

Finally, to the goal (2). First I need to explain where the object cv3 came from. Look back at the ->-intro rule which gave rise to this sub-goal. The third premise, which gave us the third sub-goal, introduces a variable x in type C which may appear free in the object in type D. Such variables are called constrained variables since they are constrained by some assumption to be in a particular type. So, cv3 is simply PICTCalc's version of a constrained variable and takes the place of x.

Going back to the derivation we can see that, again, ->-intro matches, this time with C as B, D as A, v2 as d(x). To satisfy the first two sub-goals generated by the tactic we use the same combination of `uelim`, `assumption`, `form` and `next` as before. This leaves us with a display as shown in figure 6.

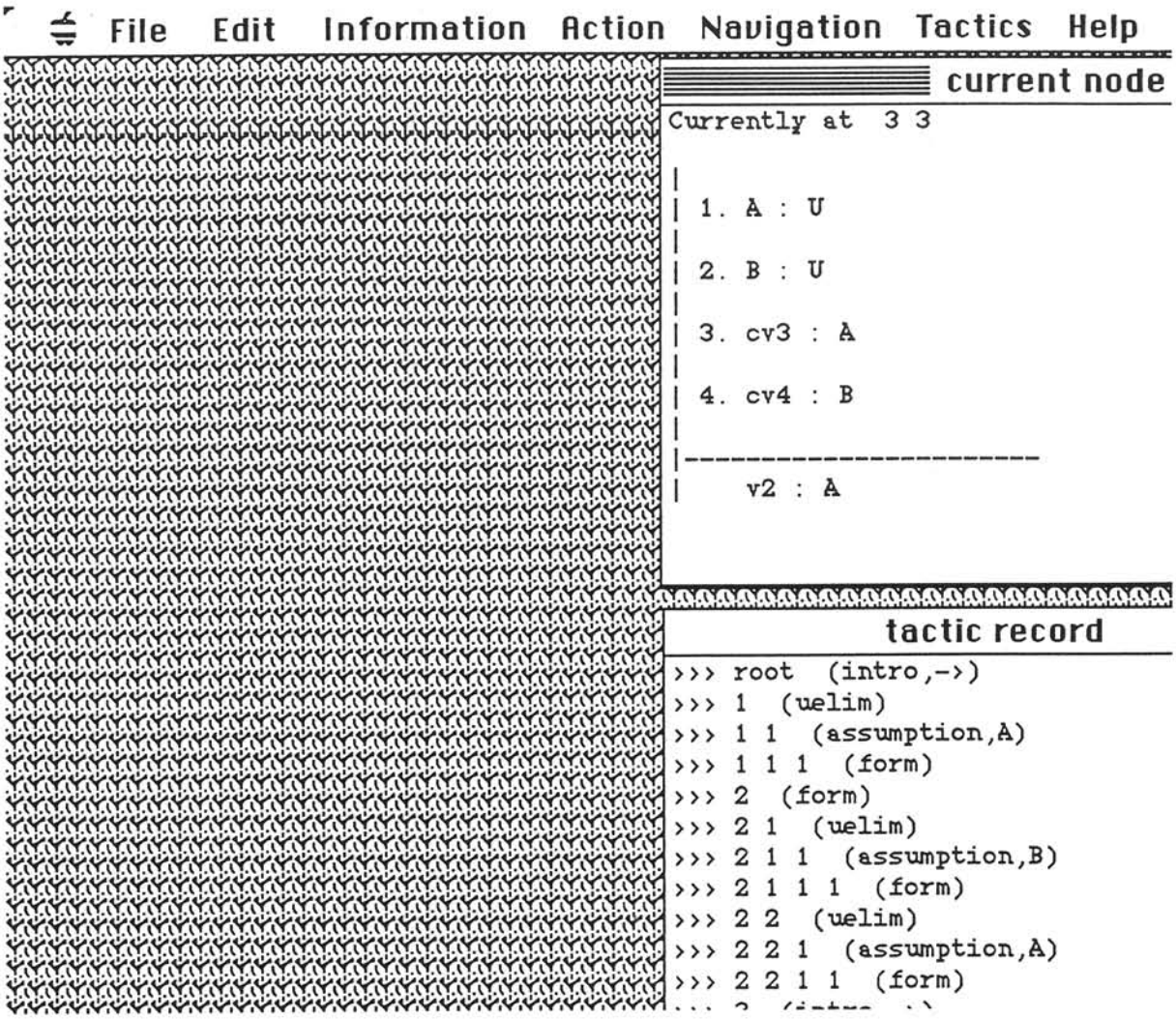


Figure 6

This is satisfied by `assumption` (which automatically instantiates the scheme variable v2 to cv3), followed by `uelim`, `assumption` and `form`.

Once all the goals are satisfied, use of `next` will result in a dialogue which tells us that there are no more sub-goals, followed by a dialogue confirming that the derivation is completed along with a display of the judgement we have just shown. Finally, the user is asked whether or not they want to save this judgement as a theorem.

Note that since A -> (B -> A) is the type of the K combinator, the proof object built should not surprise you.

In order to get more idea of what exactly has been built by PICTCalc it is a good idea at this stage to use the `Navigation` options to move around the derivation and reconstruct it on paper in its more usual two-dimensional form. If you do this for the current example you should build something which looks like the derivation shown in figure 7.

Doing a derivation "by hand" as above is very instructive since you see all of the work that is involved in finding the derivation. However, it can become very tedious after a while, especially when the derivation repeatedly involves trivial but large sub-derivations such as

showing that some expression denotes a type. For this reason there is an automatic option, `auto` in the `Action` menu (shortcut ⌘A), which follows a pre-set sequence of tactics which, time and experience have shown, can helpfully reduce the amount of tedious work that the user has to do.

In order to see this in action, `restart` the above derivation and choose `auto`. You should find that the derivation is built with far less effort on your part.

As a second example we consider trying to construct an object in the type

(A -> (B -> C)) -> ((A -> B) -> (A -> C))

assuming that A, B and C are types, i.e. with the assumption [A : U,B : U,C : U]. Note that this is the type of the S combinator.

Choose `start`, then press the `construct` button followed by the `user` button in the ensuing dialogues and give the assumption list and type as above.

This time use `auto` immediately to do the derivation. The first tactic to use is `intro`.

After a lot of work the system stops with the goal

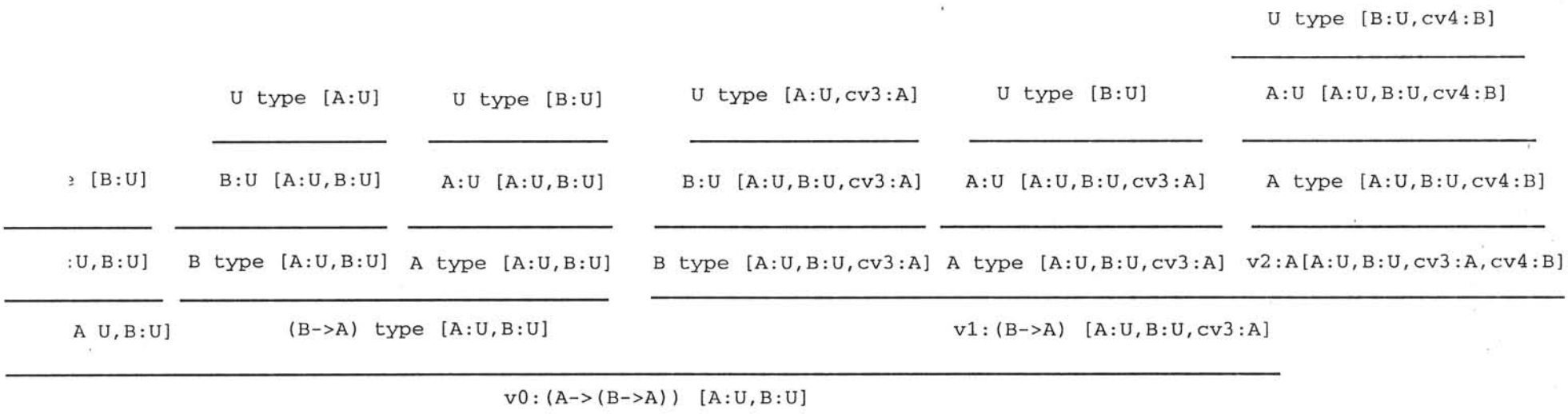
v1 : ((A -> B) -> (A -> C)) [A : U,B : U,C : U,cv4 : (A -> (B -> C))]

which again needs `intro` to deal with it, which then leads to further work and the goal

v2 : (A -> C) [A : U,B : U,C : U,cv4 : (A -> (B -> C)),cv5 : (A -> B)]

which also needs `intro`, finally leading to the goal

v3 : C [A : U,B : U,C : U,cv4 : (A -> (B -> C)),cv5 : (A -> B),cv6 : A]



bound to lambda((x1)@ lambda((x2)@ x1))

)ound to lambda((x1)@ cv3)

)ound to lambda((x1)@ cv3)

In the conclusion we clearly have all the ingredients necessary for constructing an object to put in place of v3: applying cv5 to cv6 gives us an object in B; applying cv4 to cv6 gives us an object in B -> C; applying the object just constructed which is in B to the object just constructed which is in B -> C gives us the required object in C. In order to do this part of the derivation formally, however, we need to do some more work because there is no rule which allows us to apply cv5 to cv6, giving an object in B, when the goal is to construct an object in C. What we need to do is use a rule that introduces an intermediate type of the right form.
The rule `subst` is as follows

$$\frac{a:A \qquad b(x):B(x)\ [x:A]}{b(a):B(a)}$$

and we can use it to introduce (considering the rule backwards as a tactic) a type A which is added to the assumptions of our current goal. Viewed logically `subst` is a cut rule. Given our goal

v3 : C [A : U,B : U,C : U,cv4 : (A->(B->C)),cv5 : (A->B),cv6 : A]

what we want is to have `B` and `B->C` as assumptions so that we can derive `C` by ->-elim. This requires several uses of `subst` introducing `B->C` and `B` twice each, together with some uses of `elim`.
Since this combination of `subst` and `elim` is so useful and so often needed a derived rule called `hyp` exists within PICTCalc (as it does in PICT) and is

$$\frac{d : C \text{ -> } D \qquad c : C}{ap(d,c) : D}$$

which introduces the type C. The existence of this rule means we can give an alternative derivation for the goal above. We do this by using `hyp` and introduce the type B, which leads to the sub-goals

v4 : (B -> C) [A : U,B : U,C : U,cv4 : (A -> (B -> C)),cv5 : (A -> B),cv6 : A]

v5 : B [A : U,B : U,C : U,cv4 : (A -> (B -> C)),cv5 : (A -> B),cv6 : A]

each of which is easily shown by using `elim` first on assumption 4 and then on assumption 5.
As further examples I have listed below the sequences of user inputs needed (which I call derivation scripts) when running `auto` to make derivations which show the propositions intuitionistically true by building proof objects:

Falsity -> A [A : U], i.e. anything follows from the false proposition, needs
`intro` at the root
`elim` on assumption 2 at node 3
(A -> B) -> (A -> (B \/ C)) [A : U,B : U,C : U] needs
`intro` at the root
`intro` at node 3
`intro` at node 3 3 introducing the constructor inl
`elim` at node 3 3 2

A -> ¬(¬(A)) [A : U] (note here that a space is needed between "->" and "¬" and also that "¬" is typed using option-l) with the definition ¬ ≡ x@(x -> Falsity)) needs
`intro` at the root
`unfold` at node 2 1
`unfold` at node 3
`intro` at node 3 1
`elim` at node 3 1 3 using assumption 3

(A -> (B /\ C)) -> (A -> B) [A : U,B : U,C : U] needs
`intro` at the root
`intro` at node 3
`hyp` at node 3 3 using intermediate type B /\ C
`intro` at node 3 3 1
`elim` at node 3 3 1 3 using assumption 6
`elim` at node 3 3 2 using assumption 4

5.2 Simple arithmetic

Having seen some "logical" examples we now look at some "computational" ones.
First, make a definition plus ≡ (a,b)@rec(b,a,(x,y)@succ(y)) and then consider deriving the judgement

plus(a,b) : N [a : N, b : N]

Remember that since we want here to make a derivation for this judgement (rather than construct a proof object during the construction of the derivation as in the logical examples) you need to press the `derive` button during the starting dialogues.
You will find that, using `auto`, this derivation is done completely automatically once plus is unfolded using `unfold1`.
We can do the derivation again "by hand". Choose `restart` from the `Action` menu without changing the input in the ensuing dialogues. You should find that you need use only

`unfold1`, `form`, `intro` , `assumption` and `elim` along with `next` to move to unsatisfied sub-goals when a branch of the derivation has been completed.

This last `elim` is a bit more complicated than the others. Since you simply want to eliminate the "rec" in the conclusion you should choose `conclusion` in the dialogue for `elim`.

You should also note, as mentioned above, that we get four sub-goals rather than the three you might expect by looking at the usual presentation of the rule, which has just three premises. As was said before, the reason for this is that the rules we usually see are abbreviations of the full rules as used in PICTCalc, so the full form of the N-elimination rule, as used by PICTCalc, is

$$\frac{c : N \quad C(x)\ \text{type}\ [x : N] \qquad d : C(0) \qquad e(x,y) : C(\text{succ}(x))\ \ [x : N,\ y : C(x)]}{\text{rec}(c,d,e) : C(c)}$$

where the second premise here is new and checks, essentially, that C is well-formed as a type abstraction.

When you have constructed the derivation, save the theorem under the name "plustype" so it can be used later on.

Next we show

$$(\text{plus}(b,0) = b) : N\ [b : N]$$

Again using `auto` this is shown automatically having unfolded the plus. However, as before, it is instructive to re-construct the derivation by hand without using `auto`. Save it as a theorem called "pluszero1".

Slightly harder to show is

$$(\text{plus}(0,a) = a) : N\ [a:N]$$

since we need to do induction (i.e. N-elim) on the variable a and before the judgement is in the right form to do that we need to apply eqelim to it (as you can see from the form of the rule as shown above). Once this is done, though, the derivation is not too hard, especially if you use "plustype" and "pluszero1" during the course of it. Save it as the theorem "pluszero2".

As a next example, consider

$$(\text{plus}(a,\text{succ}(b)) = \text{succ}(\text{plus}(a,b))) : N\ [a : N,\ b : N]$$

Even though it looks quite a complicated result (and perhaps you might dive in with N-elimination immediately) you will find that by using `auto` and then unfolding the "plus" it can be done with no help.

Store the result as the theorem "pluscomm1".

Next consider

$$(\text{plus}(\text{succ}(a),b) = \text{succ}(\text{plus}(a,b))) : N\ [a : N,\ b : N]$$

This innocent looking variant on the previous example is much harder and does require induction, i.e. N-elim. First do `eqelim` and then `elim` with assumption 2, which means we do the induction on "b". After some type-checking, which "plustype" via `thm` deals with, we

get to the base case of the induction. Do an `intro` and then use `substeq` with "plus(a,0)" (which has type "N"). The tactic `substeq` is based on the rule

$$\frac{(a = b) : A \qquad c(b) : C(b)}{c(a) : C(a)}$$

so it is the "a" in this rule that is prompted for by the tactic.
Now the next goal, corresponding to the first premise of the rule using a scheme variable to stand for the unknown "b" in the rule, will be

(plus(a,0) = v4) : N [a : N, b: N]

which can be solved by quoting theorem "pluszero1". After checking the type of "a" we are left with the goal

(plus(succ(a),0) = succ(v4)) : N [a : N,b : N]

which is solved by using `substeq` with "plus(succ(a),0)". This gives the goal

(plus(succ(a),0) = v5) : N [a : N,b : N]

which we use theorem "pluszero1" on and by committing to v4 and v5.
Having completed the base case we go on to the inductive step. We have to show that (after an initial use of `intro`)

(plus(succ(a),succ(cv3)) = succ(plus(a,succ(cv3))) : N
[a : N,b : N, cv3 : N, cv4 : I(N,plus(succ(a),cv3),succ(plus(a,cv3)))]

noting that the inductive hypothesis appears, with its proof object "cv4", as assumption 4. To prove this goal we use `substeq` with "plus(a,succ(cv3))", which gives the goal

(plus(a,succ(cv3)) = v6) : N
[a : N,b : N, cv3 : N, cv4 : I(N,plus(succ(a),cv3),succ(plus(a,cv3)))]

which we solve by using `unfold`. This in turn gives the goal

(plus(succ(a),succ(cv3)) = succ(v6)) : N
[a : N,b : N, cv3 : N, cv4 : I(N,plus(succ(a),cv3),succ(plus(a,cv3)))]

This we solve similarly, by using `substeq` with "plus(succ(a),succ(cv3))" then unfolding. This leaves us with

(v7 = succ(v6)) : N [a : N]

We use commit twice here, on "v7" and "v6" to give

(rec(cv5,succ(a),(x1,x2)@succ(x2)) = succ(rec(cv3,a,(x1,x2)@succ(x2)))) : N
[a : N,b : N, cv3 : N, cv4 : I(N,plus(succ(a),cv3),succ(plus(a,cv3)))]

This is now, in fact, the assumption 4. To see this we use `unfold` and then `eqelim`. `auto` will see this and use `assumption` and finish-up with some straightforward typechecking. Save this theorem as "pluscomm2".

Finally, to show how the previous theorems can be used, we show

$$(\text{plus}(a,b) = \text{plus}(b,a)) : N\ [a : N, b : N]$$

This again, of course, uses induction, and we will also see how breaking a derivation into theorems helps to make the way clearer.

First we use `eqelim` and `elim` with assumption 2. After some typechecking, using "plustype" and `thm`, this leaves us with the goal where "b" has been replaced by "0", i.e. the base case. This can be solved by using `intro` and then `substeq` using "plus(a,0)" and then the theorem "pluszero1". Next, we commit to "v4", and use `sym` followed by the theorem "pluszero2".

Next we move to the inductive step, on which we first use `intro`. We can then use `substeq` with "plus(a,succ(cv3))" followed by use of the theorem "pluscomm1". Then commit to "v5" and use `substeq` with "plus(succ(cv3),a)" followed by use of the theorem "pluscomm2". Then commit to "v6". This leaves us with a goal which is the inductive hypothesis, i.e. assumption 4. So, to finish, we use `eqelim` followed by some more typechecking. Save the theorem as "pluscomm".

In this example we see how important the three preceding derivations were because the two theorems proved in them allowed the current derivation to be much smaller and so easier to understand.

5.3 The theorem cache

There is an option within PICTCalc which, used judiciously, can further improve the construction of a derivation like the one just described. In the `Action` menu there is an option `set up cache`. This controls the size of a theorem cache, which is a structure within PICTCalc which holds the most recently used theorems. When in `auto` the system sees whether or not any of the cached theorems can be used to extend the derivation and if so uses them automatically.

To see this in action, `restart` the derivation above and set the cache size to five. The first theorem to be used, via `thm`, is "plustype" and it is placed in the cache (silently as far as the user is concerned). Later on, whenever "plustype" can be used to extend the derivation the system automatically uses `thm` and "plustype". You can see this happening by watching the "tactic record". This mechanism lessens the amount of interaction required by the user.

I said that the cache must be used judiciously and the reason for this caution is clear. There is an overhead involved with the cache because at every node it looks through the cache to see whether there is a theorem in it that can be used. Clearly, if the cache is large this checking can take a long time, so beware.

The cache is at its most useful when there are many trivial typechecking steps for which you need just to apply a theorem. In this case choose `next` at each of these, which means ignoring them at the time they appear, and then when all the non-trivial parts of a derivation are finished, set the size of the cache appropriately, load the needed theorems by using each of them once and then leave the system to work its way through the remaining, trivial nodes, using `thm` and entries from the cache as it goes.

6. Further derivation scripts for PICTCalc

6.1 Logic

((A -> B) /\ (A -> C)) -> (A -> (B /\ C)) [A : U, B : U, C : U]
`intro`
`intro`
`intro`
`hyp` using A
`elim` with 4
`hyp` using A
`elim` with 4

(A -> (B \/ C)) -> ((B -> D) -> ((C -> D) -> (A -> D))) [A : U, B : U, C : U, D : U]
`intro`
`intro`
`intro`
`intro`
`hyp` using B \/ C
`intro`
`elim` with 9
`elim` with 6
`elim` with 7
`elim` with 5

6.2 Simple computational examples

Peano's fourth axiom states that, for any n and m in N, succ(m) = succ(n) entails m = n, which in CTT can be stated as

$$(m = n) : N\ [m : N, n : N, p : I(N,succ(m),succ(n))]$$

In order to derive this judgement it turns out to be convenient to define pred ≡ n@rec(n,0,(x,y)@x), then the script is

substeq using n
inst using pred(succ(m))
substeq using succ(m)
inst using succ(n)
undo (because auto is too eager to use intro here!)
eqelim
commit to v2
unfold
commit to v1
unfold

Some obvious properties of the Booleans are:
I(Bool,true,false) -> Falsity (that is, true is not equal to false)
intro
eqtype using if(true,Falsity,N)
substeq using true
eqelim
commit to v2
eqtype using N
inst using 0

Finally, and very easily (using auto) we get:
if(true,a,b) = a : A [A : U,a : A, b : A]
if(false,a,b) = b : A [A : U,a : A, b : A]

7. Algorithm development

In this section we will look at a derivation which shows how the type language is rich enough to express both specifications and programs which meet the specifications. It is our first example to show the elegance and power of CTT. In general the language is expressive enough to denote specifications for any total, computable function and to denote the function too.
The example is taken from [NPS90], which is a good introduction (in a syntax fairly close to PICTCalc's) to all aspects of CTT.
First we need some definitions and it turns out that having

$$\text{plus} \equiv (a,b)@\text{rec}(b,a,(x,y)@\text{succ}(y))$$

and

$$\text{mult} \equiv (a,b)@\text{rec}(a,0,(x,y)@\text{plus}(b,y))$$

and

$$1 \equiv \text{succ}(0) \text{ and } 2 \equiv \text{succ}(1)$$

is adequate. What we are actually going to do is construct an object in

```
∏(N,(x1)@
        ∑(N,(x2)@
                (I(N,x1,mult(x2,2)) \/ I(N,x1,succ(mult(x2,2))))
         )
 )
```

This says that for all x1 in N, there is an x2 in N such that either x1 = x2*2 or x1 = (x2*2)+1.

For convenience we assume that

```
d : ∏(N,(x1)@
            ∏(N,(x2)@
                    (I(N,x1,succ(mult(x2,2))) -> I(N,succ(x1),mult(succ(x2),2)))
             )
    )
```

which is to say that d is a proof that, for any x1 and x2 in N,

if x1 = (x2 * 2) + 1 then x1 + 1 = (x2 + 1) * 2

which is clearly a simple fact of arithmetic.

This derivation is done in a context where was have the theorem "multx2" which simply states

mult(x,2) : N [x:N]

This is very simple but comes up so many times that it needs to be a theorem. The best way of doing the derivation is to set the cache size to zero, skip all the judgements which need the theorem, saving them until there is nothing left to do and then setting the cache size to 1, loading the cache with "multx2" by explicitly using it once and then letting the system get on with finishing the proof by several uses of the theorem. Under these circumstances the derivation script is (leaving out all uses of `next` to skip the theorem):

```
intro
elim on 2
intro
intro using inl
instn on v4 using 0
intro
unfold
thm plustype
commit v4
elim on 4
elim on 6
intro
```

```
intro using inr
instn using on v10 using cv5
intro
eqelim
commit to v10
intro
intro using inl
instn on v15 using succ(cv5)
hyp with I(N,cv3,succ(mult(cv5,2)))
gen with cv5, abstracting on all occurrences
gen with cv3, abstracting on all occurrences
set cache size to 1
thm using multx2
commit to v15
commit to v4
commit to v10
commit to v15
```

The object we construct during this derivation is

```
lambda((x1)@rec(x1,(0,inl(eq)),
                    (x2,x3)@split(x3,
                        (x4,x5)@when(x5,
                                        (x6)@(x4,inr(eq)),
                                        (x6)@ (succ(x4),inl(ap(ap(ap(d,x2),x4),x6)))
                                    )
                                )
                )
    )
```

This is a function which given an x1 in N returns a pair consisting of the whole number part of x1/2 together with a proof that the answer is correct.

For example, if this is applied to 0 then it immediately simplifies to (0,inl(eq)), i.e. the whole number part of 0/2 is 0. If it is applied to succ(0) then it simplifies to

```
split((0,inl(eq)),
        (x4,x5)@when(x5,
                        (x6)@(x4,inr(eq)),
                        (x6)@ (succ(x4),inl(ap(ap(ap(d,0),x4),x6)))
                    )
    )
```

and then to

```
when(inl(eq),
        (x6)@(0,inr(eq)),
        (x6)@ (succ(0),inl(ap(ap(ap(d,0),0),x6)))
    )
```

and finally to (0,inr(eq)), i.e. the whole number part of 1/2 is 0.
If it is applied to succ(succ(0)) then it simplifies to

```
split((0,inr(eq)),
      (x4,x5)@when(x5,
                   (x6)@(x4,inr(eq)),
                   (x6)@ (succ(x4),inl(ap(ap(ap(d,0),x4),x6)))
                  )
  )
```

and then to

```
when(inr(eq),
        (x6)@(0,inr(eq)),
        (x6)@ (succ(0),inl(ap(ap(ap(d,0),0),x6)))
    )
```

and finally to (succ(0),inl(ap(ap(ap(d,0),0),eq))), i.e. the whole number part of 2/2 is 1.
As argued in [Tho91], this is not what we would normally think of as the "div2" function since we do not usually expect to have proof information mixed in with the computational information (though in CTT of course these are the same things - it is our interpretation outside of the theory that makes this distinction between proof objects and computational objects).
In order to get a more usual object we need to change the specification into

```
Σ(N -> N,(f)@
        Π(N,(x1)@
                (I(N,x1,mult(ap(f,x1),2)) \/ I(N,x1,succ(mult(ap(f,x1),2))))
         )
 )
```

An object in this type is a pair of the form (F,P) where F is a function in N -> N which returns the whole-number result of its argument divided by 2 and P is a proof that F has that property, i.e. a proof that it meets its specification. This is clearly the more usual way of specifying a function. The reader might like to try to construct the object in this second form of the specification.

Though we will not treat it further here, it is interesting to note that a proof of the axiom of choice (which is a theorem in this version of CTT) is a function which transforms the object in the first specification to an object in the second!

8. Developments

PICTCalc, just like MacPICT before it, is still an experimental system. Currently the only difference between the two systems is that PICTCalc runs on 32-bit Macs and MacPICT does not. Since 32-bit Macs are the future, we needed to make this port so that further development is possible.

MacPICT was used for many years to support an M.Sc. course on constructive type-theory and programming at QMW, University of London (the author's previous institution). Many benefits have flowed from putting the system into such a harsh environment!

The work reported here has also benefited from a recent survey [GRB93] of programs that are intended, more or less, to support learning about or using logical ideas.

There is clearly still great scope for improvement, both at the level of the interface (from how menus and windows are used to how the syntax is presented) and at the level of how a user interacts with the system and what information the system provides for the user. All these things are constantly being changed, sometimes only to be changed back when ideas or suggestions turn out to make the system worse!

Any comments whatsoever from anyone who uses this system are most welcome.

8.1 Feedback

There is currently a large flaw in PICTCalc, considering that it is designed for teaching, and that is that there is no feedback to the user when they try to apply a tactic wrongly. That is, they choose a tactic whose conclusion does not match the current goal.

At the very least we need a dialogue box to announce this fact (the current "silence" is very bad practice according to good HCI design principles) and ideally we would have a message which gives the reasons why the conclusion and goal do not match. In order that, as a user becomes more expert, they do not get irritated by this sort of message a menu item should allow them to turn off the feedback mechanism.

8.2 Display of derivations

As far as the interface is concerned, the biggest single improvement will be to allow a derivation to be viewed in its two-dimensional form, i.e. as a tree. This gives a "global" view of the derivation, which makes understanding it easier, and also allows it to be more directly interpreted in terms of rule applications, since the rules themselves are usually viewed as nodes in a potential tree-like derivation.

Since this is clearly a large change to be made it is going to take a good deal of time to plan the best use of the graphical interface and, of course, to implement it.

8.3 Strategies

The main difference between PICTCalc and Hamilton's original version PICT is that PICTCalc does not use strategies. This is because its predecessor MacPICT was foremost intended as a teaching tool and not as a tool for developing the kind of large scale derivations for which a strategic facility would certainly be necessary. However, as PICTCalc develops we expect that much more work will be done in the area of incorporating strategies. Having used the system extensively and having watched many other people use it, I want to start introducing a few useful and clear strategies. In fact I will be re-introducing some of Alan Hamilton's originally available ones, now that I can see that they will be useful. However, I think that they should appear as an option to the user so that when people first encounter the system there are no strategies available. This is in order that the user is confronted with as little choice as possible since experience has shown that they get so overloaded that they "freeze", not knowing which choice to make.

8.4 Infix definitions

In order to make expressions more readable I want to allow a user to specify that any definitions they make may treated either as they are now or as infix or prefix operators. This will, as in Prolog, involve the user in giving the type (infix, prefix etc.) and precedence of any definiens that they want to use as an operator.

8.5 Further Automation

Further into the future, the intention is to allow PICTCalc to use a tableau-based system to construct proof objects for simple propositions. The plan is that after being initiated, either because of a direct request from the user or because PICTCalc itself needs the object, the construction will happen "in the background" and require no interaction with the user.

8.6 Finally...

PICTCalc will also be developed to run on Unix workstations under X and Motif, using experience gained on re-implementing another calculator, MiraCalc, which supports work on functional programming in Miranda.

9. Conclusions

PICTCalc, as it currently stands, has been successful in that it has provided much needed support for students learning and using a formal program development system. Of course, much remains to be done (according to the students) and the points made in the previous section arise from comments over the years.

As well as providing support it has also motivated the students by making the task of working within a formal system far more concrete; it has also made such working seem much more like "conventional" computer science in that they sit at a computer and interact with a system.

This, perhaps, sounds like something trivial and, even, harmful. However, experience shows that it is important, even for students on a formal methods M.Sc., to provide an environment which lives up to a student's expectations of what working in computer science means. It is important both for the student's own motivation but also it allows them to feel that what they are doing is central and relevant to "real" computer science; it helps to counteract the tendency to view formal computer science as irrelevant to "real" problems. It also means that people from outside the course looking in feel, for the same reasons, that the work the formal methods students are doing is relevant to computer science. It helps to make the point that work in this area is part of mainstream computer science.

Appendix

1. Information

The options in this menu give information about the current derivation.

1.1 show

Displays the current node.

1.2 addr

Gives the address of the current node.

1.3 leaves

Lists all the currently unproved leaves of the derivation below the current node.

1.4 showdef

Displays the definiens for the chosen definiendum.

1.5 showthm

Displays the judgement which is saved under the chosen name as a theorem.

1.6 showbinding

Shows the expression that the chosen scheme variable is bound to.

2. Action

The options here control the proof assistant and are concerned with starting a derivation, backtracking when a mistake is made or a change to the derivation is required. The way that theorems are used when in the "auto" mode is controlled here. Also included here are options to set up, store and remove definitions, theorems and derivations.

2.1 start

This is used to initiate a derivation. The user is asked where the input is coming from, either the `user` via the keyboard or a `file`. The file should contain a list of assumptions followed by "." and then the type or judgement (as appropriate) followed by ".". The user may intend either to `construct` an object, in which case the input consists of an assumption list and a type, or they may intend to `derive` a judgement, in which case the input will consist of a complete hypothetical judgement.

2.2 restart

This is used to repeat the initiation of a derivation. The user can either keep the inputs as before or edit them.

2.3 auto

Using this option causes PICTCalc to try each of a certain list of tactics on the current goal, passing on to any sub-goals which arise until it reaches a goal when none of the tactics on its list work. At this point the user is prompted for a tactic to use next.

2.4 undo

This undoes the most recent step, i.e. removes that last generated child nodes, and makes the relevant parent node the current node.

2.5 back

This undoes all the derivation that has been constructed (by auto) since the last user input.

2.6 chop

This removes all of the derivation above the current node.

2.7 set up cache

This sets the size of the theorem cache.

2.8 newdef

This prompts for a definiendum and a definiens and sets up a definition in the database.

2.9 newthm

To save the resulting judgement of a completed derivation in the database this option should be used. The user is prompted for a name.

2.10 loaddefs

This is used to add the definitions in a named file to those currently in the database.

2.11 loadthms

This is used to add the theorems in a named file to those currently in the database.

2.12 loadderivation

This is used to replace the current derivation (if there is one) with the derivation stored in the named file.

2.13 savedefs

This is used to store the definitions in the current database in the named file.

2.14 savethms

This is used to store the theorems in the current database in the named file.

2.15 savederivation

This is used to store the current derivation in the named file.

2.16 rmdef

This is used to remove the named definition from the database.

2.17 rmthm

This is used to remove the named theorem from the database.

2.18 quiet

If this option is ticked then nothing will be written in the tactic window. To tick it, choose it; to untick it, choose it.

3. Navigation

The options here allow the user to move around the current derivation by changing the denotation of "current node".

3.1 up

Use this to move up to the left-most child node of the current node.

3.2 down

Use this to move down to the parent node of the current node.

3.3 left

Use this option to move to the next node to the left at the current level.

3.4 right

Use this option to move to the next node to the right at the current level.

3.5 move

This moves us to the node with the specified address.

3.6 toroot

This moves us to the root of the derivation.

3.7 next

This moves us to the next unproved node.

4. Tactics

This menu contains all the rules that can be used to extend the current derivation.

4.1 form

An appropriate formation rule is used to extend the derivation.

4.2 intro

An appropriate introduction rule is used to extend the derivation.

4.3 elim

An appropriate elimination rule is used to extend the derivation. This option displays some buttons which allow the user to say where the elimination should be done, i.e. in one of the assumptions or, if the conclusion is of the form a : A or (a1 = a2) : A where a, a1 and a2 are not canonical, in the conclusion. Any further information that might subsequently be needed is prompted for if necessary.

4.4 hyp

If the current goal is of the form

$$vn : B$$

then this option uses the rule

$$\frac{a : A \qquad b : A \to B}{ap(b,a) : B}$$

i.e. ->-elim, backwards and prompts for the intermediate type, i.e. A, giving rise to the appropriate sub-goals.

4.5 gen

If the current goal is of the form

$$vn : B$$

then this option uses the rule

$$\frac{a : A \qquad b(x) : B(x)\ [x : A]}{ap(b,a) : B(a)}$$

i.e. $\prod$-elim, backwards and prompts for some sub-expression a of B(a), from which it either calculates the type A and the abstraction B or asks for them if it cannot.

4.6 uelim

The universe elimination rule is used to extend the derivation.

4.7 comp

If the current goal is of the form

$$(a = b) : A$$

then this option applies the appropriate computation rule, if there is one. If b is a scheme variable it is instantiated by use of the rule. Other information may be prompted for if required.

4.8 assumption

This option extends the derivation by using the assumption rule.

4.9 refl

This uses the reflexivity of =.

4.10 sym

This uses the symmetry of =.

4.11 trans

This uses the transitivity of =. It introduces a new scheme variable as the unknown, central term.

4.12 subst

This extends the derivation by using an appropriate substitution rule. If the goal has the form

$$b(a) : B(a)$$

then the rule used is

$$\frac{a : A \qquad b(x) : B(x)\ [x : A]}{b(a) : B(a)}$$

and the user is prompted for a (and possibly its type if that cannot be calculated).
If the goal has the form

$$(b(a) = c(a)) : B(a)$$

then the rule used is

$$\frac{a = d : A \qquad b(x) = c(x) : B(x)\ [x : A]}{(b(a) = c(d)) : B(a)}$$

and the user is prompted for objects a and d (and possibly their types).
If the goal has the form

$$B(a)\ \text{type}$$

then the rule used is

$$\frac{a : A \qquad B(x)\ \text{type}\ [x : A]}{B(a)\ \text{type}}$$

where a is prompted for.

If the goal has the form

$$B(a) = C(d)$$

then the rule used is

$$\frac{(a = d) : A \qquad B(x) = C(x)\ [x : A]}{B(a) = C(d)}$$

and a and d are prompted for.

4.13 substeq

This implements a derived rule, namely

$$\frac{(a = c) : A \qquad J(c)}{J(a)}$$

where J(a) is any form of judgement where a may appear free. Here a is prompted for and c is a scheme variable invented by the system.

4.14 eqelim

This extends the derivation by use of the I-elim rule. If the goal has the form

$$(a = b) : A$$

and b is a scheme variable then if there is an assumption of the form

$$e : I(A,a,c)$$

then b is instantiated to c and the sub-goal

$$(a = c) : A$$

extends the derivation.

If no such suitable assumption exists or b is not a scheme variable then the sub-goal extending the derivation is simply

$$v : I(A,a,b)$$

for a new scheme variable v.

4.15 eqcomp

A goal of the form

$$(a = b) : A$$

where b is a scheme variable leads to a sub-goal of the form

$$(a = eq) : A$$

4.16 eqtype

This extends the derivation by using one of the rules for equality of types.

4.17 commit

This is used to commit to an instantiation of a scheme variable made at any previous stage. When using auto, the user is asked whether to commit to any previously instantiated scheme variable that appears in the current node before anything else is done.

4.18 inst

This extends the derivation by allowing the instantiation of the scheme variable v in goals of the form

$$v : A \quad \text{or} \quad (v = b) : A \quad \text{or} \quad (a = v) : A$$

The user is prompted for an expression to instantiate v with and the derivation is extend by adding the appropriate form of sub-goal in each case.

4.19 instn

This extends the derivation by allowing the instantiation of a scheme variable anywhere in the current goal. The user is asked which of the scheme variables in the current goal should be instantiated and what expression it should be instantiated with.

4.20 thm

Using this option extends the derivation by taking a theorem, given by the user, from the database and, having calculated an appropriate instance of it, adding its assumptions as sub-goals of the current node.

4.21 unfold

This unfolds all definienda in the current goal, replacing them by the appropriate definiens in each case.

4.22 unfold1

This unfolds a chosen definiendum in the current goal.

5. Help

This menu contains, essentially, the information given in the previous four sections, arranged by option name.

References

[Bee85] Michael J. Beeson, "Foundations of Constructive Mathematics", Springer-Verlag, 1985.

[Bit92] Oliver Bittel, "The λ-Tableau Calculus: A New Approach to Theorem Proving in Intuitionistic Logic", extended abstract in "Theorem Proving with Analytic Tableaux", internal report 8/92, Institute for Logic, Complexity and Deduction Systems, University of Karlsruhe, Germany.

[GRB93] Doug Goldson, Steve Reeves and Richard Bornat, "A Review of Several Programs for the Teaching of Logic", The Computer Journal, July 1993.

[Ham92] Alan G. Hamilton, "The PICT Guide", Department of Computing Science, Technical report, TR 99, University of Stirling, December 1992.

[M-L84] P. Martin-Löf, "Intuitionistic Type Theory", Bibliopolis, 1984.

[NPS90] Bengt Nordström, Kent Petersson, Jan Smith, "Programming in Martin-Löf's Type Theory: An Introduction", Oxford, 1990.

[Tho91] Simon Thompson, "Type Theory and Functional Programming", Addison-Wesley, 1991.