[1]Yet Another Introduction to Constructive Type Theory

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

Abstract

In this paper we present, very briefly, the background to CTT and introduce some of the proof rules that formalize it. We then do some small examples to show how the rules are used and look at two more general points: using a proof of the axiom of choice as a specification transformer and showing how working constructively can mean that you use information during program construction that, classically, would not be available.

1. Introduction

There has been much written, and there is much to say, about the ideas behind Martin-Löf's Constructive Type Theory (or Constructive Set Theory) [M-L84]. If we move away from the particular formal system that is CTT to the underlying philosophy and mathematics (intuitionism) upon which it is based there is even more to say, and even more people have said it [Bee85, Dum77].

Rather than rehearse all of the arguments for and against intuitionism and for and against CTT, in this paper we will take the formalisation as it is and see some of what we can do with it. Having said that, however, a few points will be made in the Conclusion.

If you have never seen CTT before, you are going to find even these small examples hard to understand (I guarantee). However, there are many accessible introductions, with [M-L84,Bee85,NPS90,Ree91,Tho91] being some examples.

1.1 Some jargon and definitions

Given that there are many ways of presenting CTT, we need to start with some jargon. The descriptions given here of the machinery of the formal system are meant just to introduce some of the words we will use; they are not meant as definitions.

---

[1]This paper appears in the Proceedings of the First New Zealand Formal Program Development Colloquium, working paper 94/18, University of Waikato, NZ.

CTT works in the natural deduction style. The basic aim, given a specification S, is to build a derivation with a judgement of the form e : S at its root. Such a derivation will have the form

$$\dfrac{\dfrac{\overset{....}{J11}\;\;A11\ldots\overset{....}{J1n1}\;\;A1n1}{J1\quad A1}\qquad \dfrac{\overset{....}{J21}\;\;A21\ldots\overset{....}{J2n2}\;\;A2n2}{J2\quad A2}\;\ldots\ldots\;\dfrac{\overset{....}{Jk1}\;\;Ak1\ldots\overset{....}{Jknk}\;\;Aknk}{Jk\quad Ak}}{e\;:\;S\qquad A}$$

where each of the Jis is a judgement and each of the Ais (and A) is a list of assumptions, which are themselves judgements. The Ji/Ai pair are called the premises of the rule and e : S/A is called the conclusion of the rule. We say that the judgement e : S depends on (or has) the assumptions in A. e : S follows from J1...Jn because of a rule, which will have the form

$$\dfrac{\mathcal{J}1\quad \mathcal{A}1\;\ldots\;\mathcal{J}k\quad \mathcal{A}k}{e\;:\;\mathcal{S}\quad \mathcal{A}}$$

where J1, A1,...,Jk, Ak, e, S and A are instances of $\mathcal{J}1$, $\mathcal{A}1$,...,$\mathcal{J}k$, $\mathcal{A}k$, e, $\mathcal{S}$ and $\mathcal{A}$ respectively.

J1 then follows from J11 A11...J1n1 A1n1 because of a certain rule and J11 is similarly produced by some rule, and so on up the derivation (which has the form of a tree of rule instances). The leaves of the derivation will be judgements which need no further rules to justify them. Such a rule is

$$\dfrac{}{N\ \mathrm{type}}$$

which simply says that N is a type without further justification (as we will see later). Note that 'A type' for certain A is another sort of judgement.

In the judgement e : S, e is an element of S (when S is viewed as a type or set), or an object (or program) meeting the specification S or (viewing S logically) e is a proof of the proposition S. ("Types are Specifications", "Propositions are Types" and "Proofs are Programs" are all (true) slogans of CTT). All these readings are equivalent - they simply represent different ways of interpreting the formal object e.

Sometimes it is convenient to introduce abbreviations for certain complicated terms. We shall write

$$d \equiv f$$

to mean that d is an abbreviation for f.

Finally, since 'λ' has a particular meaning within the formal system, and since we want to use the usual idea of abstraction and application (that the λ-calculus expresses) in the machinery of CTT, we need some alternative notation (not involving 'λ') to denote abstractions and applications of expressions. To this end we write

(x)e

to denote the expression that, unofficially, we might be tempted to write as λx.e. So we have that (x)e(a) is the same as e with all free occurrences of x in e replaced by a (with renaming if needed to avoid 'variable capture' as usual). We say that (x)e is an abstraction and that e(a) is an application. The expression e(a) can therefore be viewed either as an expression containing some free occurrences of a or as some abstraction e followed by '(', followed by some expression a, followed by ')'. That is to say the expressions (x)x+2(3) and 3+2 are the same. Also, we allow abstractions on tuples of variables, so (x,y)x+y(1,2) and 1+2 are the same expression, and repeated abstraction, so (x)(y)x+y(2)(3) and 2+3 are the same.

2. Some proof rules

We have already seen one proof rule, namely the one that says that N is a type. Another simple and useful rule is

$$\frac{A \text{ type}}{x : A \; [x : A]} \text{ assumption}$$

which says that if A has been shown to be a type then we can introduce the variable x as having that type, by assumption. Note how this assumption is recorded in the list after the judgement.

Other rules involving N are

$$\frac{a : N}{\mathrm{succ}(a) : N} \text{ N-intro-succ} \qquad\qquad \frac{}{0 : N} \text{ N-intro-0}$$

$$\frac{n : N \quad d : C(0) \quad e(x,y) : C(\mathrm{succ}(x)) \; [x : N,\, y : C(x)]}{\mathrm{rec}(n,d,e) : C(n)} \text{ N-elim}$$

Taking the first two, we can see that derivations like the following are easily constructed

$$\frac{\dfrac{}{0 : N}}{\mathrm{succ}(0) : N} \qquad\qquad \frac{\dfrac{\dfrac{}{0 : N}}{\mathrm{succ}(0) : N}}{\mathrm{succ}(\mathrm{succ}(0)) : N}$$

Taking the last rule, we can construct the derivation

$$\dfrac{\dfrac{}{0 : N} \qquad \dfrac{}{0 : N} \qquad \dfrac{\dfrac{N\ \text{type}}{x : N\ [x : N]}}{\text{succ}(x) : N\ [x : N]}}{\text{rec}(0,0,(x,y)\text{succ}(y)) : N} \qquad \text{(Here C is the abstraction (a)N)}$$

This last rule, N-elim, formalizes induction over the natural numbers within CTT. The first premise introduces an arbitrary n in N. The second premise says that we must have some proof, d, of some arbitrary proposition C(0), i.e. some proposition possibly involving 0. The third says that we have some proof e(x,y) of the proposition C(succ(x)), i.e. some proposition possibly involving succ(x) in the same way as 0 was involved in the previous premise, where we can assume that x is in N and that y is a proof that C(x). This third premise, then, is what we would informally call the inductive step and the assumption introducing y is the inductive hypothesis. Given all of these we can conclude that C(n), i.e. the proposition introduced in the premises is true of the arbitrary element n, is proved (with a rather complicated looking proof).

So far, we have nothing which looks like computation. A rule involving N that does look computational (and which explains the proof in the N-elim rule) is

$$\dfrac{d : C(0) \qquad e(x,y) : C(\text{succ}(x))\ [x : N,\ y : C(x)]}{\text{rec}(0,d,e) = d : C(0)}\ \text{N-comp-0}$$

and another is

$$\dfrac{n : N \qquad d : C(0) \qquad e(x,y) : C(\text{succ}(x))\ [x : N,\ y : C(x)]}{\text{rec}(\text{succ}(n),d,e) = e(n,\text{rec}(n,d,e)) : C(n)}\ \text{N-comp-succ}$$

The reader can check that, using these rules, we can derive

rec(0,0,(x,y)succ(y)) = 0 : N,

rec(succ(0),0,(x,y)succ(y)) = succ(0) : N

and

rec(succ(0),succ(0),(x,y)succ(y)) = succ(succ(0)) : N.

In fact, the rules above go towards making sure that the type N behaves just like the natural numbers, so 'succ' forms successors, 0 acts as zero and 'rec(a,b,(x,y)succ(y))' acts like a + b whenever a and b are elements of N.

3. Deriving a simple numerical algorithm

The example used in this section is taken from [NPS90] and was treated in a similar manner in [Ree94], where a fuller explanation and presentation of the derivation, using the calculator MacPICT (or “PICTCalc” as it is now called) can be found.

First we need some definitions:

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

and

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

and

$$1 \equiv \mathrm{succ}(0)$$

and

$$2 \equiv \mathrm{succ}(1)$$

What we are going to do is construct an object in

$$\prod(N,(x1)\sum(N,(x2)(I(N,x1,\mathrm{mult}(x2,2)) \vee I(N,x1,\mathrm{succ}(\mathrm{mult}(x2,2))))))$$

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

$$d : \prod(N,(x1)\prod(N,(x2)(I(N,x1,\mathrm{succ}(\mathrm{mult}(x2,2))) \rightarrow I(N,\mathrm{succ}(x1),\mathrm{mult}(\mathrm{succ}(x2),2)))))$$

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.

If we follow through the derivation (which we won't do here, see [Ree94]) then the object we construct is

$$\begin{aligned}&\lambda((x1)\mathrm{rec}(x1,(0,\mathrm{inl}(eq)),\ (x2,x3)\mathrm{split}(x3,(x4,x5)\mathrm{when}(x5,(x6)(x4,\mathrm{inr}(eq)),\\ &\qquad\qquad (x6)(\mathrm{succ}(x4),\mathrm{inl}(d(x2,x4,x6)))))))\end{aligned}$$

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

$$\mathrm{split}((0,\mathrm{inl}(eq)),(x4,x5)\mathrm{when}(x5,\ (x6)(x4,\mathrm{inr}(eq)),(x6)\ (\mathrm{succ}(x4),\mathrm{inl}(d(0,x4,x6)))))$$

and then to

$$\mathrm{when}(\mathrm{inl}(eq),(x6)(0,\mathrm{inr}(eq)),(x6)(\mathrm{succ}(0),\mathrm{inl}(d(0,0,x6))))$$

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(d(0,x4,x6)))))

and then to

when(inr(eq),(x6)(0,inr(eq)),(x6)(succ(0),inl(d(0,0,x6))))

and finally to

(succ(0),inl(d(0,0,eq)))

i.e. the whole number part of 2/2 is 1.

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). We consider this point further in the next section.

4. Transforming the specification

The form of the specification above was

∏(N,(x1)∑(N,(x2)e(x1,x2)))

so that we got a function which, for any number x1, gives a pair consisting of the number x1/2 and a proof that this was correct. That is, we get something like

0 -----> (0, a proof that 0/2 = 0)
1 -----> (0, a proof that 1/2 = 0)
2 -----> (1, a proof that 2/2 = 1)

and so on.

However, it is more usual to expect the specification to yield a pair consisting of a function, which works for all numbers, and a proof that the function is correct when we ask for a solution to the 'div2' problem. That is, we get

$(f, \lambda n.\text{proof that } n/2 = f(n))$

In order to get a more usual object we need to change the specification into

$$\Sigma(N \rightarrow N,(f)\Pi(N,(x1)(I(N,x1,mult(f(x1),2)) \vee I(N,x1,succ(mult(f(x1),2))))))$$

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.

Now we turn to a seemingly unrelated (and surprising) problem, that of proving the axiom of choice. It turns out that the axiom of choice is neither independent of nor inconsistent with CTT - it is in fact a theorem of CTT. This has to do with particular properties of the formalisation we are considering, and Martin-Löf himself seems to have been unsure whether this property was desirable or not. We shall not worry about such things here, and simply carry on.

The axiom of choice (in one of its many equivalent forms) says that if given any family of sets we can collect together into a set one element from each member of the family such that some condition holds then there is a function to do it. (This is clearly not interesting if the family is finite - it becomes interesting when the family is infinite.)

We can formalize this statement by using A as the set which indexes the family, B(x) as a member of the family whenever x is from A, and C is the condition which the element from B(x) satisfies.

So, we have

$$\Pi(A,(a)\Sigma(B(a),(b)C(a,b))) \rightarrow \Sigma(\Pi(A,B),(f)\Pi(A,(a)C(a,f(a))))$$

It is not too hard to show that

$$\lambda((x1)\ (\lambda((x2)fst(x1(x2))),\lambda((x2)snd(x1(x2)))))$$

is an element of this type, so we have proved the axiom of choice. Further, and more interestingly for us, consider the antecedent to the axiom of choice and compare it with the specification for the div2 problem. Note that they match, with A as N, B as (x)N and C as (I(N,x1,mult(x2,2)) ∨ I(N,x1,succ(mult(x2,2)))). So, we would expect that applying the proof of the axiom of choice to the element in the div2 specification would construct for us an element in the corresponding consequent of the axiom of choice, i.e. an element in

$$\Sigma(N \rightarrow N,(f)\Pi(N,(a)(I(N,a,mult(f(a),2)) \vee I(N,a,succ(mult(f(a),2))))$$

(using the abbreviation $\Pi(N,(x)N)$ is $N \rightarrow N$).

We can check that this is so:

applying the proof of the axiom of choice to the 'div2' object gives the pair

(λ((x2)fst(rec(x2,(0,inl(eq)),(x2,x3)split(x3,(x4,x5)when(x5,(x6)(x4,inr(eq)),(x6)
(succ(x4),inl(d(x2,x4,x6)))))),
λ((x2)snd(rec(x2,(0,inl(eq)),(x2,x3)split(x3,(x4,x5)when(x5,(x6)(x4,inr(eq)),(x6)
(succ(x4),inl(d(x2,x4,x6)))))))

Moving the fst and snd in as far as possible gives

(λ((x2)rec(x2,0,(x2,x3)split(x3,(x4,x5)when(x5,(x6)x4,(x6)succ(x4))))),
λ((x2)rec(x2,inl(eq),(x2,x3)split(x3,(x4,x5)when(x5,(x6)inr(eq),(x6)inl(d(x2,x4,x6)))))))

The first element is

λ((x2)rec(x2,0, (x2,x3)split(x3,(x4,x5)when(x5,(x6)x4,(x6)succ(x4)))))

which is a function in N → N which returns the whole number part of x2/2

The second element is

λ((x2)rec(x2,inl(eq),(x2,x3)split(x3,(x4,x5)when(x5,(x6)inr(eq),(x6) inl(d(x2,x4,x6))))))

which is a function which given x2 returns a proof that the function in the first part gives the right answer.

5. On using all available information

This next example is considered informally and says something rather general about algorithm specification and derivation - something which is not specific to CTT. Consider the function called 'lambo' (and invented by Lambeck and Moser - this section is based on some lecture material by Samson Abramsky). Given any function $f : N \to N$ such that f is unbounded we have

lambo f n = the least m such that $f(m) \geq n$

lambo f n is easily computed by

```
i:=0;
while f(i) < n do
      i:=i+1
deliver i
```

However, the interesting question asks whether we can specify this function lambo in CTT. The problem seems to be that, since f is unbounded, lambo must necessarily embody some unbounded search, which CTT cannot do.

We start by asking for the type of lambo. The standard Milner typing would be

$$\text{lambo} : (N \to N) \to (N \to N)$$

However, with the far more expressive CTT we can do much better. In particular, we must not forget that the first argument to lambo must be an unbounded function. To say that this first argument, f, must be unbounded is to say that for all n in N, there is an m in N such that $f(m) \geq n$. In CTT this is

$$\Pi(N,(n)\Sigma(N,(m)(f(m) \geq n)))$$

where we assume that we have $\geq$ such that $a \geq b$ is inhabited iff a is greater than b.

Given this we can write a specification for lambo

$$\begin{array}{l}\Pi(\Sigma(N \to N, (f)\Pi(N,(n)\Sigma(N,(m)(f(m) \geq n)))), \\ \qquad (g)\Pi(N, (p)\Sigma(N, (q)(\text{fst } g\ q \geq p \wedge \Pi(N, (r)(\text{fst } g\ r \geq p \ \to\ q \leq r))))))\end{array}$$

So, note that lambo takes as its first argument a <u>pair</u>: the first element of the pair is the function f of type $N \to N$ which is unbounded and the second element, when applied to any number n, returns a pair consisting of a number m and a proof that $f(m) \geq n$.

This greater care about the type of lambo, then, has given us some more information which allows us to compute lambo using a bounded search. The fact that f is unbounded allows us to calculate the upper bound needed for our search. The computation for lambo can now be written as

```
lambo (f,h) n  =  for i:=0 to fst h(n) do
                      if f(i) ≥ n then return i
```

and so lambo could be derived within CTT.

6. Conclusion

CTT provides us with a single language (syntax, semantics and proof theory) within which we can specify and implement programs. Of course, this still leaves us with the problem of knowing whether or not the specification is correct. However, this will always be a problem until someone invents a formal system in which people will naturally think and wish to express and solve their computational problems and it which it will be as obvious as it currently is, when such 'requirements' are expressed in a natural language like English, that someone has asked for exactly what they want.

Given that we work in a single language, we do at least not have the problem of taking a specification language and an implementation language and trying to 'join' then together via some logic.

The logic that the proof rules express is constructive, which is to say that whenever we prove an existential statement we always construct the object whose existence is proved and whenever we prove a disjunction we always know which of the disjuncts holds. It is hard to see how a logic that deals with computation could be otherwise, without some elaborate apparatus to sort out the constructive from the non-constructive parts of the theorems proved.

One mode of attack on such an idea as CTT is to express incredulity that someone would ever expect to work completely within a formal system. Of course, such attacks are empty - no one does ever work completely within a formal system. Just like the formal notations (i.e. notations that look more mathematical than natural, e.g. Z or VDM, and whose semantics and proof theory are usually left informal or at least not used explicitly within the specification process) a formal system like CTT provides a means for being precise and expressing facts which might have been arrived at, initially, by any means whatsoever. What the proof theory in CTT gives us, in addition, is the means of checking these facts within the system itself. Again, the checking might initially be done informally and then expressed within CTT in order to both feed into the implementation process and also to check the checking itself.

The syntax of CTT is not attractive or useable. The first of these points is highly subjective and the second is untrue. However, the particular syntax used is just a convention and is not the most important part of the system. It certainly reflects the mathematical and logical background of its inventor, but could easily be changed.

As with any other formal system which deals with large formal objects (e.g. any programming language, specification language, proof checker, theorem prover) we run into problems with CTT as we become more ambitious. The more mundane problems of editing, keeping up-to-date versions of, documenting and generally interacting with large formal objects are probably more important (and in some ways harder to solve) than the mathematical or logical problems that CTT and its like deal with.

Solving these sorts of large-scale problems requires experience and skills from many different areas within Computer Science - Software Engineering, HCI and Logic are the obvious ones. Finding people with these skills (or putting together groups with these skills) is hard, given the specialization that we are all forced to undergo as part of our training. Overcoming the limitations of our specialization is the real problem.

References

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

[Dum77] Dummett, M., "Elements of Intuitionism", Oxford University Press, 1977.

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

[NPS90] Nordström, B., Petersson, K. and Smith, J. "Programming in Martin-Löf's Type Theory: An Introduction", Oxford, 1990.

[Ree91] Reeves, S., “Programming as Constructive mathematics”, in “Mathematical Structures for Software Engineering”, ed. De Neumann, Simpson and Slater, Oxford, 1991.

[Ree94] Reeves, S., “Computer support for students' work in a formal system: MacPICT” to appear in the *International Journal on Mathematical Education in Science and Technology.*

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