A New Formulation

of

Constructive Type Theory

David Turner
University of Kent
England

PLAN

1) Locat
and i(
(unde

2) Sketc
overc

3) Some
trans

[Note - (
develop
language

PLAN

1) Locate constructive type theory and identify a problem (undecidability of well-typing)

2) Sketch of new theory overcoming problem

3) Some remarks about referential transparency

[Note - underlying motivation is to develop practical programming language based on ctt]

ory

The constructive theory of types (Martin-Löf 1982) is a collection of <u>judgements</u>

$$p : P$$

(main form of judgement, there are others)

Admits of two readings

1) proof : PROPOSITION

2) element : TYPE

Elements are always computable, so CTT is also a programming language

Example of a judgement

(U_0 is universe of types, alias propositions)

$$\lambda(A:U_0)\lambda(x:A)\, x \;:\; \forall(A:U_0)\, A \rightarrow A$$

This can be read as either

1) a typing for the polymorphic identity function

or

2) a proof of the (2nd order) proposition that every first order proposition implies itself

CTT is two things simultaneously

1) A system of (intuitionist) logic with a notation for proofs as well as propositions

2) A strongly typed functional programming language with the unusual property that all programs terminate.
(= hereditary totality)

CTT is a theory of a fundamentally new kind, providing a single integrated framework for programs and proofs.

The isomorphism between propositions and types

CTT is based on a discovery made by H. B. Curry in 1958, of a funny coincidence between simple typed lambda calculus and (intuitionist) propositional calculus.

Illustration:-

$A \rightarrow A$ is a tautology

$A \rightarrow B$ is not a tautology

exists a closed λ expression of type $A \rightarrow A$

not exists closed λ expression of type $A \rightarrow B$

DOES THIS ALWAYS WORK ?

YES ! !

Howard showed that this can be extended to include all the connectives of (intuitionist) propositional calculus.

<u>connective</u>	<u>logic reading</u>	<u>type reading</u>
$\rightarrow$	implication	function space
&	conjunction	cartesian product
$\vee$	disjunction	disjoint union
$\perp$	absurdity	empty type

<u>**RESULT**</u> (Curry/Howard circa 1960)
F is a tautology of intuitionist propositional calculus <u>iff</u> there is a closed λ expression whose type can be written as **F**

(there is a second result relating normalisability to cut-elimination)

The original Curry/Howard isomorphism is between simple typed λ calculus and intuitionist propositional calculus, but it is part of a much more general relationship between logic and programming.

CTT of Martin-Löf extends the isomorphism to include both quantifiers, and higher order forms (CTT is an ω order logic).

In fact the Curry/Howard isomorphism is the root of a whole family of theories of which CTT is only one.

Some of the theories based on the Curry Howard isomorphism

[1] Howard 1980

"The formulae-as-types notion of construction" in "To H B Curry, Essays on formalism" Academic Press 1980
THE ORIGINAL THEORY OF CURRY AND HOWARD

[2] Martin-Löf 1973

"An intuitionist theory of types - predicative part" in Logic Colloquium 73 (North Holland)
AN EARLIER VERSION OF CTT

[3] Coquand and Huet 1985

"A Theory of Constructions" International Symposium on Semantics of Data Types, Sophia Antipolis, 1985.
A NEW, IMPREDICATIVE THEORY

[4] Martin-Löf 1982

"Constructive mathematics and computer programming" reprinted in Mathematical Logic & Programming Languages, Prentice Hall 1985.
THE NOW STANDARD VERSION OF CTT

Actually [4] is <u>very odd</u>, compared with the others.

Theories [1], [2], [3] and others all have:-

- The : judgement is decidable
- Unicity of type
- Strongly Church-Rosser (strong normalisability)

BUT all these theories lack an extensional concept of equality.

e.g. cannot prove within the theory

$$\lambda(x{:}N)\ x + 1 \;=\; \lambda(x{:}N)\ 1 + x$$

[4] alone has extensional equality

$$\frac{\forall x \,.\, f\,x = g\,x}{f = g}$$

this is a fundamental requirement for reasoning about functions - Per Martin-Löf abandoned the earlier version of CTT because it lacked this rule.

BUT in [4] (the now standard version of CTT)

- The : judgement is undecidable
- Unicity of type is lost
- Normal forms do not exist (in general)

THE PRICE OF EXTENSIONALITY ?

NEW THEORY [Turner 88, full account in preparation] has

- The : judgement is decidable
- Unicity of type
- Strongly Church-Rosser
- Extensional equality

It is actually rather close to the old version of CTT [Martin-Löf 73], but introduces extensional equality by a different method from that used in [Martin-Löf 82]

The key step is, equality is a <u>proposition only</u>, not a judgement.

NTT (= new type theory) is a collection of closed sentences of the form

$$p : P$$

This is the sole form of judgement - it has the usual two readings

The type operators are

$$\forall \quad \exists \quad \vee \quad N_k \quad N \quad W \quad U_i \quad =$$

All bound variables are annotated with their types (requires some extra type info to be inserted)

Judgements are generated by using two sorts of rules

- Natural deduction rules written

  premise $\vdash$ conclusion

  or equivalently

$$\frac{\text{premise}}{\text{conclusion}}$$

- Computation rules written **a** ->> **a**'
  e.g. β, η reduction rules

->> is strongly Church-Rosser and preserves validity of judgement (on both sides of ":")

[Aside:-
so there is a decidable relation <<->> of <u>definitional equivalence</u>]

UNIVERSES

U_0 is collection of all (first order) propositions or types.

There is a hierarchy of universes
U_0, U_1, U_2, ... (ramified theory)

Each type belongs to exactly one universe. Examples

$$N : U_0$$

$$(N \to N) \to ((N \to N) \to (N \to N)) : U_0$$

$$U_0 \to U_0 : U_1$$

The rule of universe formation is

[U_iF] $\quad U_i : U_{i+1} \quad i \geq 0$

∀ formation

If A is a type and B is a family of types containing zero or more free occurrences of a variable x of type A, then $\forall(x{:}A)\,B$
is the type of dependent functions from A to B. As proposition means universally quantified statement

$$[\forall F] \qquad \frac{A{:}U_i\ ,\ x{:}A \vdash B{:}U_j}{\forall(x{:}A)B : U_{max(i,j)}}$$

If B does not contain x free, then

$$\forall(x{:}A)B$$

can be abbreviated to

$$A \rightarrow B$$

the usual function type, which as proposition means implication.

∀ introduction

$$\frac{x : A \vdash b : B}{\lambda(x{:}A)b : \forall(x{:}A)B}$$

∀ elimination

$$\frac{f : \forall(x{:}A)B \;,\; a : A}{f\ a : B[a/x]}$$

∀ computation rules

[β] $(\lambda(x{:}A)b)\ a \;\twoheadrightarrow\; b[a/x]$

[η] $\lambda(x{:}A)f\ x \;\twoheadrightarrow\; f$

(if x not free in f)

∃ formation

If A is a type and B a type formula containing zero or more free occurrences of a variable x of type A, then $\exists(x:A)B$ is the type of dependent pairs $(a,b)_{\exists(x:A)B}$
As proposition means existential.

$$[\exists F] \qquad \frac{A:U_i\ ,\ x:A \vdash B:U_j}{\exists(x:A)B : U_{max(i,j)}}$$

If x not free in B then

$$\exists(x:A)B$$

may be abbreviated to

$$A \mathbin{\&} B$$

the ordinary pair type.

As proposition means conjunction.

∃ introduction

$$\frac{a:A\ ,\ \ b : B[a/x]}{(a,b)_{\exists(x:A)B} : \exists(x:A)B}$$

∃ elimination rules

1) $$\frac{p : \exists(x:A)B}{fst(p) : A}$$

2) $$\frac{p : \exists(x:A)B}{snd(p) : B[fst(p)/x]}$$

∃ computation rules

$$fst(a,b)_T \ \ \text{->>}\ \ a$$

$$snd(a,b)_T \ \ \text{->>}\ \ b$$

Also straightforward, and essentially the same as CTT (except for some extra type witnessing)

A v B disjunction alias disjoint union
(introduces conditional branching)

$\mathbf{N_k}$ finite type with k members
(introduces case switch)
special case $\mathbf{N_0}$ is absurdity alias the empty type

N natural numbers
(introduces primitive recursion alias mathematical induction)

W(x:A)B general inductive type
gives arbitrary well-founded trees and <u>transfinite recursion</u>

Equality formation

$a = b$ is proposition expressing the thought that a, b denote the same abstract object. Not well formed unless a, b have the same type.

Since we have unicity of type, we do not need to write $a =_A b$
an important difference from CTT

$$[=F] \qquad \frac{a:A\ ,\ b:A\ ,\ A:U_i}{a = b\ :\ U_i}$$

sing the
ne <u>same</u>
formed
ype.

ype, we

m CTT

<u>= introduction rules</u>

1)
$$\frac{a : A}{\mathrm{selfid}(a) : a = a}$$

2)
$$\frac{e : \forall(x{:}A)\, b_1 = b_2}{\mathrm{ext}(e) : \lambda(x{:}A)b_1 = \lambda(x{:}A)b_2}$$

<u>= elimination</u> (slightly simplified)

$$\frac{e : a = b \;,\; p : P[a/x]}{\mathrm{subst}(e,P,p) : P[b/x]}$$

This is Leibnitz's law
<u>aka</u> the rule of referential
transparency

Equality computation rules

1) ext(λ(x:A)selfid(b))
   ->> selfid(λ(x:A)b)

2) subst(selfid(a),P,p) ->> p

These are needed to ensure that there is only one proof that an expression is equal to itself

Example of an equality proof

```
ext(λ(n:N)
      primrec(n,
                 selfid(0),
                 <ind>
                )
   ) : λ(n:N)n+0 = λ(n:N)n
```

<ind> is a proof of the proposition

$$n+0 = n \rightarrow suc(n)+0 = suc(n)$$

here omitted for brevity.

Note that in CTT the whole proof would be written as just

$$r \; : \; \lambda(n:N)n+0 = \lambda(n:N)n$$

In CTT every true equation has the same proof, namely the atom **r**.
In NTT equality proofs have an internal structure recording how the equation was proved.

A proof of the induction step

```
λ(e:n+0=n)
  subst(e,
          λ(z:N)suc(z)=suc(n),
          selfid(suc(n))
        )
    :  n+0=n → suc(n)+0=suc(n)
```

on step

(n),

suc(n)

overall comparison

CTT:-

4 forms of judgement
8 type constructors
128 rules of inference
judgement <u>undecidable</u>

NTT:-

1 form of judgement
8 type constructors
32 rules of inference
judgement <u>decidable</u>

The key advantage of NTT is the existence of a decision procedure for well typing

NTT has a finer type structure, so less judgements are valid.
However, I believe that the propositional consequences are <u>essentially the same</u> (i.e. no loss of power in going from NTT to CTT)

Remarks on referential transparency

We are familiar with the principle of referential transparency - that in a proposition we may substitute equals for equals

$$\frac{a = b\ ,\ \ P[a]}{P[b]}$$

However in an intuitionist theory we have not only propositions P, but also JUDGEMENTS

$$p : P$$

Question: Does the principle of ref. transparency apply to judgements also?

According to Martin-Löf's CTT, "yes".

But is this right?

<u>How referential transparency works in NTT</u>

Suppose we have a proof **e** of **a=b** and suppose we have for some predicate **P**, a proof of **P[a]**

$$p \ : \ P[a]$$

then by equality elimination we get

$$\mathrm{subst}(e,P,p) \ : \ P[b]$$

So <u>propositions</u> are referentially transparent - but judgements are not, for we do NOT (usually) have

$$p \ : \ P[b]$$

So in NTT judgements are referentially opaque

[Compare - in CTT judgement is referentially transparent. This makes judgement undecidable, because <u>equality is undecidable</u>]

What, technically, is the difference between proposition and judgement?

My claim is that there are just two rules to follow

1) judgement is decidable

2) propositions are ref. transparent

When mathematics reaches the level that equality becomes undecidable (functions as values) it follows inevitably that

1a) judgement is ref. opaque

2a) propositions are undecidable

CLAIM:- CTT went wrong because Martin-Löf tried to keep judgement referentially transparent while introducing an extensional equality

SUMMARY OF NTT POSITION

Propositions substitutive under =
but judgements not (because proofs are not).

However, judgements are fully substitutive under <<->>, which is a kind of "syntactic equality"

[Explanation:-

A proposition is a relationship between abstract objects.

A judgement is a relationship between syntactic objects.

]

Discussion after David Turner's talk

David Turner: Ok, that turns out to be more complicated than I first thought. One computation rule you must have obviously is that a *Subst* and a *Selfid* cancel out. If you have a *Selfid* in the e-position in $Subst(e, P, p)$ ... But you want to know what happens if there is an x here? It gets quite complicated. It turns out, you have to read inside and do actual substitution, so you have to have rules for pushing *Subst* through formulae, which is not unreasonable because ... when you compute with it, you end up doing substitutions. It adds a lot of extra rules because for every constant, you have to have a rule that says how you push a *Subst* through it, i.e. push a *Subst* through a *cons* etc.

Per Martin-Löf: I just want to say about these equality rules: this is the same identity rule as I use.

David Turner: Yes, I know.

Per Martin-Löf: And I have an elimination operator which is more general than this *Subst* and which contains *Subst* as a special case. They will compute, if I remember it correctly, the way you've just said. The real important point is this one, the extensionality rule.

David Turner: Right, that's the thing which is different.

P. Martin-Löf: What you want is apparently what I would call an extensional version of type theory which has at least this extensionality axiom because you may think of other extensions I suppose in addition to this one.

David Turner: Yes, for example, if you want quotient types then more things will come in.

Per Martin-Löf: Then my attitude is that you can make perfectly good sense of these axioms, but you will do that in a way which is analogous to what I think Gandy was the first to give: an interpretation of extensional simple type theory into the intensional version of simple type theory. And Takeuti did later and independently. Something similar to that can also be done for my type theory i.e. you can formulate an extensional version of type theory and make sense of it by giving a formal interpretation into the intensional version.

David Turner: Will that mean that the extensional version will be consistent if the intensional version is?

Per Martin-Löf: Yes, and much more by making precise sense to those extensional equalities. In that sense you may succeed in making sense of this. But I don't think you can claim that it is obvious as it stands.

David Turner: It depends on what you think a function is. If you think a function is a rule or method than it is not obviously true, if you think a function is charactarized by its graph then you want that to be true.

Per Martin-Löf: Yes, you want that to be true and so you must clarify this notion of a function as a graph and that's precisely what this axiom does.

David Turner: In the end there is a practical reason. Why I want to think of functions as characterized by their graphs is that if I write a functional program and there are two ways to define a certain function and say one of them is more efficient than the other, so I develope my program using f. I later want to prove that f is extensionally equal to g. I want a general principle that tells me I can unplug f and plug in g in the program and it will still be right, no matter what the program was doing with f, whether it was applying it or passing it as a parameter or proving things about it. So that's why I for quite practical reasons want to have a rule like this. Because that's one appeal of functional programming, that you can code a function in two different ways and know that they are interchangable in all contexts.

Per Martin-Löf: Two possibilities: either work within the intensional theory and prove that the particular context in which you want to make the replacement is actually extensional or else work all the time in an extensional theory but then of course you must remember that the meaning of everything is rather indirect. You must convince yourself of the validity of the axioms.

David Turner: Yes, I must be an uninformed classical thinker and deep down inside I believe that functions are characterized by their graphs. What I want to say is: if you don't know this rule then you don't know what a function is. If we don't have this rule we are not talking about functions, we are talking about algorithms.

Per Martin-Löf: Rather I think you should say that if you haven't seen that rule, you don't know what extensional equality between functions means. It's part of the nature of being a function, that the appropriate kind of equality between functions is extensional.

N.G. de Brujin: Completely independent of this question, you should be aware of the fact that what you tell here is exactly AUTOMATH in 1968 - and at that moment we knew very well that otherwise the thing would not be decidable. Well we had some options, we needed to write all these things as axioms. But I think in this form it was written up in Jutting's version of Landau.

David Turner: I can believe that you would have the same proof rules, but you wouldn't presumably have the isomorphism between programs and proofs.

N.G. de Brujin: Well, one was not talking about programs at that moment – but mathematics. We still had that possibility, that's no problem – but two different equalities: the definitional equality and ...

David Turner: ... you had that distinction ...

N.G. de Brujin: ... and these equalities were only introduced in the book, they were not in the language-definition at all. In the book you could choose this treatment, you could also do it in other ways.

David Turner: Which, if any, of the equalities was built into the language?

N.G. de Brujin: Definitional equality.

David Turner: OK and then you just defined the extensional equality from it.

N.G. de Brujin: You can define it, you can also take it as an axiom.

David Turner: So this is more like the position in Martin-Löf's system that extensional equality is not basical.

N.G. de Brujin: Martin-Löf did not take that over and you have just pushed it back again.

Gérard Huet: Don't you want in your logic of programs to be able to write a statement such as Quicksort is a better algorithm than Bubblesort?

David Turner: I believe that Quicksort is a better algorithm than Bubblesort but I'm not sure that's the sort of thing you want to say in the logic. That's part of the complexity theory. The logic is just going to tell you that they compute the same function and that is what you want to know. Logic is not supposed to answer questions about efficiency, is it?

Bengt Nordström: When I see the requirements on your judgements, it's like the things to the left of the epsilon are derivations.

David Turner: I suppose it is. The proof object recapitulates the derivation. I think that's part of the Curry-Howard isomorphism. This is true in the theory of constructions as well and it's true in the original theory of Howard.

Bengt Nordström: But if you have that view, this extensionality requirement doesn't make sense.

David Turner: It's a requirement on propositions.

Bengt Nordström: But you're treating two functions as equal if they are extensionally equal and the two functions are functions between derivations ...

David Turner: ... Are you saying that it is inconsistent?

Bengt Nordström: No, no. I'm just saying that your motivation behind this extensionality doesn't seem to fit with your view of elements as derivations.

David Turner: ... There are judgements which are about syntax and there are judgements which are about abstract objects, and there's a different kind of equality in the two rules and different ways referential transparency is working in the two rules.

N.G. de Brujin: You can still use the definitional equality of functions as a notion and work on that and prove theorems about it ... anyway, that's how we descibe algorithms in AUTOMATH.

David Turner: Presumably we all agree that there is an interesting relation on functions, stronger than definitional equality, which is kept as extensional. What we call that, and how we formalize it, is a different question, but we've got to talk about both things.

Bengt Nordström: But my point is that the extensionality view of functions is most interesting when you treat functions as programs, I mean, your argument about substituting functional programs ...

David Turner: Extensionally equal functions are not interchangable in proofs – I see what you're saying, they're denoting programs. But any proposition I can make about a program is also true about the substituted program ... If f and g are extensionally equal, I can't replace f by g on the left of the colon, because, if it's a proof, it's like replacing $sin^2x + cos^2x = 1$ by $1 = 1$, which is a silly thing to do. But any proposition I could make about f, I can make about g, so in particular, if this program using f is correct, I can make substitution in that and this program using g is correct. The substitution I want to do is actually on the right hand side of the colon ... I want to make propositions about my programs and know they are invariant under substitution of extensionally equal functions.

Peter Aczel: You have focused on talking about things on the right hand side of the colon being thought of as propositions, but of course, they are also to be thought of as types. What's your story about referential transparency. Suppose you have an element in a dependent type, depending on some function, and you replace it by an extensionally equal function, then the element is no longer in the new type. What do you want to say about that?

David Turner: I think what I say is that it is a special case of this : I may have $a : A$ and $A = B$, but I'm not going to get $a : B$ ... My kind of types is not extensional because equal types don't have the same members. What's true is there's going to be an $\hat{a} : B$. So they're isomorphic. But they don't behave extensionally and that's why I think, for example, it would be quite wrong to call them sets. They're not types. Membership is intensional, not extensional.

Per Martin-Löf: What do you mean by the equality there, $A = B$?

David Turner: It's generated by the substitution rules from the equality on terms.

Per Martin-Löf: So it just means that, in set theoretic terms, if one is nonempty then the other is nonempty and vice versa.

David Turner: No, it means more than that. For example, it will mean that there's a bijection between their members. How you get it is ... So that's a good reason for not writing that sign as epsilon ... don't behave like extensional collections.

Michael Hedberg: You said once that your theory has only one form of judgement, the membership, but computation must take part in formulating the rules, so actually there are more judgement forms.

David Turner: Well, I tend to think of the computation arrow as an auxiliary form of judgement ...

Michael Hedberg: But could you formulate the rules of the theory without using the computation arrow?

David Turner: No, I don't think so, no. Not enough things would belong to each other.

Jan Smith: A little comment on the comparision with Martin-Löfs 82-theory. I think this is not really the extensional equality you have there, because that was a very strong rule in the sense that it confuses judgements and propositions.

David Turner: And made membership extensional as well.

Jan Smith: And together with universes that's something very strong, so I doubt your conjecture that you could prove the same things.

David Turner: Let me tell you what the conjecture is. Let me leave out the universes. What I think is true is that if $a : A$ in the 82-theory, then there exists a' and A' so that $a' : A'$ in my theory with $A = A'$ and $a = a' : A$ in the 82-theory. This is without universes ... I don't know how to raise this to the first universe.

Jan Smith: I don't think it holds.

David Turner: You think it actually doesn't hold. Why?

Jan Smith: There happens a lot of strange things. For instance, with a universe and these strong equality rules, you may even write down programs which have nonterminating parts.

David Turner: I can't do that. So the claim that the theories have the same propositional consequences is true only inside the first universe.