An Introduction to Constructive Type Theory in Qi II

Mark Tarver
Lambda Associates
dr.mtarver@ukonline.co.uk

Author's Note: this documentation is a transcript of a series of 7 posts on the subject of applying Qi II to constructive type theory made to Qilang in May 2009. The text has been very lightly edited. The structure and informal style of the original posts has been retained and the limit of each post has been marked with an underline. The system described here can be downloaded from www.lambdassociates.org/Lib/libraries.htm (see Logic section)

Some time back there was some discussion with respect to Qi and Coq and such. I wrote

But the basic idea is that in CTT (constructive type theory), rather than have a program which is then type checked, you have a type and then find a program to fit. In other words, a problem is less of the form 'Find a type for the following expression E', more of 'Here is a type T, find an expression that inhabits it'.

*Now generally the latter question is not very useful for programmers. However it turns out that if you feed your type theory steroids, you can cause it to beef up to the point where your type T *expresses a formal specification of what the program is supposed to do.* The catch however is that your 'roided-up' type theory is combinatorially uncontrollable. You cannot simply release your type checker on it. So you have to interact with the type checker to derive the solution. At that point you have to build a proof assistant and so you have your NuPrls and Coqs etc.*

I thought I might show you how to do CTT in Qi in a series of occasional posts. You need to load in the proof assistant in Qi Programs/Chap15 that comes in the Qi download. Its documented in http://www.lambdassociates.org/Book/page311.htm.

If we're not bothered overmuch by type security we can build a fast implementation using rule abstractions (see http://www.lambdassociates.org/Book/page306.htm and after for how to work them). A rule abstraction works on a list of ordered pairs (sequent) <A, C> where A is a list of assumption and B a conclusion.

An Introduction to the Typed Lambda Calculus

Generally a starting point is the typed lambda calculus which is part of Qi type theory and just about every typed language. I'm going to be brutal here to save space by saying anything is a wff.

Note: The definition of wff is liable to be filled in a later version

```
Qi II 2008,
Copyright (C) 2001-2008
Mark Tarver www.lambdassociates.org
version 1.06

(0-) (datatype wff

         ______
         X : wff;)
wff
```

```
(1-) (synonyms sequent ([wff] * wff))
synonyms

(2-) (tc +)
true

(3+) (define abs
        {[sequent] --> [sequent]}
        S -> ((rule X : A >> Y : B;
              ______________________
              (/. X Y) : (A --> B);) S))
abs : ((list ((list wff) * wff)) --> (list ((list wff) * wff)))

(4+) (define app
       {[sequent] --> [sequent]}
        S -> ((rule X : (A --> B); Y : A;
              _____________________
              (X Y) : B;) S))
app : ((list ((list wff) * wff)) --> (list ((list wff) * wff)))

(5+) (define hyp
       {[sequent] --> [sequent]}
       S -> ((rule ______
                   P >> P;) S))
hyp : ((list ((list wff) * wff)) --> (list ((list wff) * wff)))
```

Note: /. Is being used as the ASCII substitute for λ and --> for →.

Here's a slightly odd def. You'll see why its useful in a bit.

```
(6+) (define answer
       {[sequent] --> [sequent]}
        [(@p A C) | S] -> [(@p A C) (@p [C | A] C) | S])
answer : ((list ((list wff) * wff)) --> (list ((list wff) * wff)))
```

Load the proof assistant

```
(7+) (load "../Qi Programs/Chap15/proof assistant.qi")
....................
loaded : symbol
```

Off we go.

```
(8+) (proof-assistant _)

Input assumptions?  (y/n) n

Enter conclusion: [[/. x x] : [a --> a]]
==============================
Step 1         unsolved 1

?- [[/. x x] |:| [a --> a]]

Tactic: abs
==============================
Step 2         unsolved 1

?- [x |:| a]
1. [x |:| a]

Tactic: hyp
```

```
Real time: 7.59375 sec.
Run time: 0.0 sec.
Space: 35360 Bytes
proved : symbol
```

The proof signature of this proof is: >> [[/. x x] |:| [a --> a]], abs, hyp

OK so far? Now rule closures use unification. So we can drive the proof in different ways. We can ask.

What is the type of [/. x x]?

What function inhabits [a --> a]? The latter question brings us closer to CTT. Lets do it.

```
(9+) (proof-assistant _)

Input assumptions?  (y/n) n

Enter conclusion: [What? : [a --> a]]

==============================
Step 1          unsolved 1
?- [What? |:| [a --> a]]

Tactic: answer
==============================
Step 2          unsolved 2

?- [What? |:| [a --> a]]

Tactic: abs
==============================
Step 3          unsolved 2

?- [#:X16870 |:| a]
1. [#:X16867 |:| a]

Tactic: hyp
==============================
Step 4          unsolved 1

?- [[/. #:X16867 #:X16867] |:| [a --> a]]
1. [[/. #:X16867 #:X16867] |:| [a --> a]]

Tactic: hyp

Real time: 17.640625 sec.
Run time: 0.015625 sec.
Space: 76452 Bytes
proved : symbol
```

The final sequent gives the answer. Any legal substitution for the variables in [/. #:X16867 #:X16867] has type [a --> a]. The purpose of the answer function was to allow this information to be displayed at the end of the proof. We could also ask our program to find a most general type for [/. x x]. This ability to synthesise a program (= lambda function) from a type is the very beginning of CTT. I leave you to play with this.

end of post 1

The Curry-Howard Correspondence

This second part deals with the Curry-Howard correspondence or programs as proofs. We can sum this correspondence up in two identities.

Proofs = Programs
Propositions = Types

The glue holding these identities together comes from Heyting's semantics for intuitionistic logic. Lets take a bite out of that.

According to Heyting, we explain the semantics of a mathematical proposition by giving the conditions under which it is provable. We assume that we understand for atomic propositions like 2 x 3 = 6 how they are proved (by calculation). Heyting explains the sense of logical connectives by giving their *proof conditions*.

To illustrate; this is Heyting

> A proof of (P → Q) (P implies Q) is a function that maps any proof of P to a proof of Q.

Now go back to lambda calculus. Remember (/. X X) : (A --> A) which is a theorem of simply typed lambda calculus? It turns out that we can see --> as Heyting implication. So the type is a proposition. Under the correspondence the resulting affirmation is

(/. X X) is a proof of (A --> A)

which is true by Heyting if

(/. X X) is a function that maps any proof of A to a proof of A.

That's dead on. Because the identity function does just that!

Now the thing is that the Curry-Howard correspondence is not too exciting at this level because the simply typed lambda calculus corresponds to an extremely limited bit of propositional calculus called *purely implicational logic*. It gets interesting when you add all the other connectives. Of which, since I have to dash, more anon.

end of post 2

Last time we got to the edge of discussing a more powerful type theory based on the Curry-Howard correspondence, but using all the connectives of logic and not just -->. Lets do that.

Representing & (and)

Right; lets tackle &. The rules for & in CTT derive from Heyting.

> A proof of (P & Q) is a pair <a,b> where a is a proof of P and b is a proof of Q.

That's simple enough. There are three sequent rules that derive from this rule.

```
(define &-right
   {[sequent] --> [sequent]}
    S -> ((rule A : P; B : Q;
                ______________
                (pair A B) : (P & Q);) S))
```

```
(define &-left1
   {[sequent] --> [sequent]}
    S -> ((rule A : (P & Q);
                ______________
                (first A) : P;) S))
```

```
(define &-left2
   {[sequent] --> [sequent]}
    S -> ((rule A : (P & Q);
                ______________
                (second A) : Q;) S))
```

We would also add rules

```
(first (pair A B)) ==> A
(second (pair A B)) ==> B
```

Note something important. The type theory is already becoming intractable since the left rules can be applied ad infinitum to any problem of the form X : p where X is an unbound variable and p is any type (proposition). Hence the query What? : (A --> A) will cause an infinite regress using Qi-type T* inferencing techniques. And it gets worse!

Representing v (Or)

A proof of (P v Q) is either

1. a pair <A,0> where A is a proof of P or ..
2. a pair <B,1> where B is a proof of Q

i.e. a proof of (P v Q) is either a proof of P or a proof of Q together with a flag indicating which one is intended to be proved.

```
(define v-right1
  {[sequent] --> [sequent]}
   S -> ((rule A : P;
               ______________________
               (pair A 0) : (P v Q);) S))
```

```
(define v-right2
  {[sequent] --> [sequent]}
   S -> ((rule B : Q;
               ______________________
               (pair B 1) : (P v Q);) S))
```

The v right rule is a bit hairier. In essence it says

```
(P v Q);
(P --> R);
(Q --> R);
__________
R;
```

The complexity comes from the fact that types are involved. The rule
is

```
(define v-left
 {[sequent] --> [sequent]}
  S -> ((rule A : (P v Q);
             B : (P --> R);
             C : (Q --> R);
             ____________________
             (cases A B C) : R;) S))
```

cases is a function of course. The Qi definition is

```
(define cases
 [pair A* 0] B C -> (B A*)
 [pair A* 1] B C -> (C A*))
```

Right what does this mean?

It says that we can prove R if we can prove (P v Q); (P --> R); (Q --> R). This bit is clear. The cases function just takes the bits and assembles the proof of R from the proofs of (P v Q); (P --> R); (Q --> R).

Recall that a proof of (P v Q) is either a proof of P or a proof of Q. So the cases function will take as its first argument either a proof of P or a proof of Q plus a flag indicating which is proved. If the flag is 0, so its a proof of P, then by applying B (a proof of (P --> R)) to the embedded proof A* of P we get a proof of R. If the flag is 1, by applying C (a proof of (Q --> R)) to the embedded proof A* of Q we get a proof of R.

Notice one thing. If a proof of (P v Q) is either a proof of P or a proof of Q, what about (P v (~ P)); the Law of the Excluded Middle? Actually LEM is not a theorem in this logic since neither P nor (~ P) may be provable in a formal system.

~ (not)

After v, ~ (not) is really easy. (~ P) is just thought of as meaning (P --> void) where void is some obviously false proposition (like 1=0). i.e. think Lisp (NOT P) = (IF P NIL T). void is just CTT's NIL.

```
(define ~-right
 {[sequent] --> [sequent]}
  S -> ((rule A : (P --> void);
             _________________
             A : (~ P);) S))
```

```
(define ~-left
  {[sequent] --> [sequent]}
   S -> ((rule A : (P --> void) >> B : Q;
              ____________________________
              A : (~ P) >> B : Q;) S))
```

Proof by Contradiction

Last bit; in logic you can prove any proposition from an absurdity. For our purposes an absurdity is a proof of void.

```
(define ad-absurdum
  {[sequent] --> [sequent]}
   S -> ((rule A : void;
              __________
              (abort A) : P;) S))
```

The abort function simply raises an error.

```
(define abort
  {A --> B}
   X -> (error "~A is a proof of void!~%" X))
```

--> (implies)

We've covered this bit; its part of simply typed lambda calculus. But we'll give these rules appropriate names.

```
(define -->-right
        {[sequent] --> [sequent]}
        S -> ((rule X : A >> Y : B;
                     _____________
                    (/. X Y) : (A --> B);) S))

(define -->-left
       {[sequent] --> [sequent]}
        S -> ((rule X : (A --> B); Y : A;
                        ________
                       (X Y) : B;)              S))

(define hyp
       {[sequent] --> [sequent]}
       S -> ((rule ______
                   P >> P;) S))
```

This covers the main part of the propositional part of CTT; more precisely, the propositional part of a system called TT_0 from Per Martin-Löf. Next I'll show some theorems from this in Qi II.

end of post 3

Here are some theorems of TT_0.

```
(19+) (proof-assistant _)

Input assumptions?  (y/n) y

1. [a : [p & q]]

Input assumptions?  (y/n) n

Enter conclusion: [What? : [q & p]]
```

I'm assuming a is a proof of [p & q] and I'm asking if there is a proof of [q & p].

```
Step 1          unsolved 1

?- [What? |:| [q & p]]

1. [a |:| [p & q]]

Tactic: answer
===============================
Step 2          unsolved 2

?- [What? |:| [q & p]]

1. [a |:| [p & q]]

Tactic: &-right
===============================
Step 3          unsolved 3

?- [#:X17642 |:| q]

1. [a |:| [p & q]]

Tactic: &-left2
===============================
Step 4          unsolved 3

?- [#:X17654 |:| [#:X17655 & q]]

1. [a |:| [p & q]]

Tactic: hyp
===============================
Step 5          unsolved 2

?- [#:X17645 |:| p]

1. [a |:| [p & q]]

Tactic: &-left1
```

```
==============================
Step 6          unsolved 2

?- [#:X17671 |:| [p & #:X17673]]

1. [a |:| [p & q]]

Tactic: hyp
==============================
Step 7          unsolved 1

?- [[[pair [second a]] [first a]] |:| [q & p]]

1. [[[pair [second a]] [first a]] |:| [q & p]]
2. [a |:| [p & q]]

Tactic: hyp
Real time: 94.0625 sec.
Run time: 0.015625 sec.
Space: 162044 Bytes
GC: 1, GC time: 0.015625 sec.
proved : symbol
```

So the answer is [[pair [second a]] [first a]] (a curried form) is proof of [q & p]. That is right of course! Since a proof of [p & q] is a pair, then swapping the elements of the pair gives a proof of [q & p].

Proof signatures are a space saving device for representing proofs. I summarise the above by

```
[a : [p & q]] >> [What? : [q & p]];                    \the problem\
answer, &-right, &-left2, hyp, &-left1, hyp, hyp        \the steps\
What? = [[pair [second a]] [first a]]                   \the solution\
```

I could show you more theorems; but I won't take away your fun. Here's a few exercises for you to try. Find the solutions.

```
1. [a : [p v q]] >> [What? : [q v p]];
2. [a : [~ p]], [b : [p v q]]  >> [What? : q];
3. >> [What? : [q --> [p --> q]]]
```

Also to round off

4. Implement the structural rules for TT_0 in type secure Qi II

a. Weakening; so you can remove an assumption when you don't want it.
b. Reordering; so you can swap the positions of two assumptions.

____end of post 4

We move now to the quantifier rules for TT0. Again Heyting is the master key.

The Quantifier Rules: universal

Back to Heyting again.

A proof of (forall X : A P) is a function that, for any object O of type A, produces a proof of PO/X (the result of replacing all occurrences of X in P by O).

Thats not hard. We have two rules.

```
(define forall-left
  {A --> [sequent] --> [sequent]}
   O S -> ((rule let Q (subst O X P)
                F : (forall X : A P) >> O : A;
                (F O) : Q, F : (forall X : A P) >> R;
                ______________________________
                F : (forall X : A P) >> R;) S))
```

The first rule says that if we have a universally quantified assumption then we can instantiate it with an object O provided we can prove that O is of the right type. That's easy enough.

The forall-right rule is trickier. Let's put in an intuitive rendering.

```
(define forall-right
  {[sequent] --> [sequent]}
   S -> ((rule let O (newsym a)
               let Q (subst O X P)
               O : A >> (F O) : Q;
               ______________________
               F : (forall X : A P);) S))
```

It says we can show F is a proof of (forall X : A P) if we can prove that (F O) is a proof of PO/X where O is an arbitrary object of type A. This is sound enough. It just says that to prove a universally quantified proposition you prove it for an arbitrary case.

Now this formulation is correct, but it turns out that its not necessarily the best to use. The problem is that the (F O) : Q; often turns out to be difficult to solve. Why? Because you can end up with problems of the form

a : P >> (F a) : p where F is a variable

which you cannot solve because 'a' and '(F a)' do not unify. If we bound 'F' to say '(/. X X)' we would have.

a : p >> ((/. X X) a) : p

and then beta reduction would give a : p >> a : p

Now this course involves using something called 'higher-order unification' which factors in lambda calculus transformations into unification. But we want to keep things simple; so we look at a way of reformulating 'forall-right'.

Now the F part of the rule stands for a function i.e. something of the form (/. Y Z). So we can write.

```
(define forall-right
  {[sequent] --> [sequent]}
   S -> ((rule let O (newsym a)
            let Q (subst O X P)
            O : A >> ((/. Y Z) O) : Q;
            ________________________________
            (/. Y Z) : (forall X : A P);) S))
```

((/. Y Z) O) is an application; so beta reduction is possible. Let Beta be the result of performing this reduction. What is the relation between (/. Y Z) and Beta? Well (/. Y Z) is just the result of replacing O in Beta by some fresh variable Y and wrapping (/. Y ...) round the whole thing. Lets call this process 'abstraction' and we represent it as (abs O Z). abstraction is, in a sense, the inverse of beta reduction.

e.g. (abs a (/. X a)) = (/. Y (/. X Y)).

Now putting this into our revised version we get

```
(define forall-right
  {[sequent] --> [sequent]}
   S -> ((rule let O (newsym a)
            let Q (subst O X P)
            O : A >> Z : Q;
            ________________________________
            (abs O Z) : (forall X : A P);) S))
```

To flash ahead; what will happen is then when we synthesise a lambda expression at the end of the proof we may find lots of 'abs' in it. All we have to do to eliminate them is to restore the missing abstractions where we are told to. That's easy. The nice thing is that we avoid complexities.

The Quantifier Rules: existential

OK; this bit will look hairy. Relax your heartbeat; I'm going to decode it straight away.

A proof of (exists X : A P) is a pair <O, B> where O : A and B : PO/x

Arrgh! Read : as 'is a proof of' as well as 'is of type'. Translated out of geek speak this reads

A proof of 'there exists an X of type A such that P' is a pair <O, B> where O is of type A and B is a proof of the proposition that results from replacing all occurences of X in P by O.

So actually its not too hard. It just says that a proof of an existence claim is a pair composed of an actual object of the right type (called a *witness*) together with a proof that the object has that property.

One thing to note is that systems based on Heyting's logic require existence proofs to be constructive (hence 'Constructive Type Theory'). A constructive existence proof requires you to prove the existence of something by providing or constructing a witness. In classical logic, one can prove the existence of fs by proving that (~ (forall x (~ f x))). However in CTT, (~ (forall x (~ (f x)))) --> (exists x (f x)) is not a theorem.

When we translate Heyting into sequent calculus we get 2 rules. One right rule and one left.

Lets take the right rule first. Again I'm going to present it and then comment on it.

```
(define exists-right
  {A --> [sequent] --> [sequent]}
   Witness S -> ((rule let Q (subst Witness X P)
                       Witness : B; A : Q;
                       _______________________________
                       (pair Witness A) : (exists X : B P);) S))
```

Ok; this is not too hard, but some comments.

1. Notice that this rule is parameterised. It begins 'Witness S -> .....' The Witness is the witness (no marks for guessing that ;)).
2. Notice we put a side condition in. The motive is obvious. It constructs the instantiation of the existential proposition. However this is important because the use of 'subst' is an indicator that we expect to find some significant structure to play with. This means that we cannot, as we did at the outset, drive our formalisation in different ways by writing types and asking for propositions which they prove (remember (/. X X) : What?). So we have sacrificed flexibility in using a side condition.

Could we avoid this? Yes; possibly. what we could do is to absorb the side condition into the premises of the sequent as in

```
(define exists-right
  {A --> [sequent] --> [sequent]}
   Witness S -> ((rule Witness : B; (Q = (subst Witness X P)); A : Q;
                       ________________________________________________
                       (pair Witness A) : (exists X : B P);) S))
```

We would retain flexibility but at the cost of making the system more complex because we would have to include rules for = and subst and manage this ourselves. So we have a choice point here, and since this is a post, we'll choose the simpler more restrictive route.

The left rule states that if we have an existential assumption, we can replace it by two assumptions (relating to the fact that the inhabitant of the existential type is a pair). The first element of a proof of an existential (exists X : B P) is an object (a witness) of type B. The second is a proof of the proposition that results from replacing the bound variable by the witness.

```
(define exists-left
  {[sequent] --> [sequent]}
   S -> ((rule let P_fstA/X (subst [first A] X P)
                (first A) : B, (second A) : P_fstA/X >> R;
                ______________________________________
                A : (exists X : B P) >> R;) S))
```

Try it for Yourself!

Again you can try out these rules by pasting them into Qi (you're probably assembling quite a collection by now). Try these theorems and find the solutions for What?.

```
a : q, p : (f a), c : (forall x : q ((f x) --> (g x))) >> What? : (g a)
a : (some x : q (f x)), c : (forall x : q ((f x) --> (g x))) >> What? : (some x : q (g x))
a : (some x : q (f x)) >> What? : (some y : q (f y))
```

To remind you of the extra rules; here they are

```
(define forall-left
 {A --> [sequent] --> [sequent]}
  O S -> ((rule let Q (subst O X P)
              F : (forall X : A P) >> O : A;
              (F O) : Q, F : (forall X : A P) >> R;
              ____________________________________
              F : (forall X : A P) >> R;) S))

(define forall-right
 {[sequent] --> [sequent]}
  S -> ((rule let O (newsym a)
             let Q (subst O X P)
             O : A >> Z : Q;
             ___________________________________
             (abs O Z) : (forall X : A P);) S))

(define exists-right
 {A --> [sequent] --> [sequent]}
  Witness S -> ((rule let Q (subst Witness X P)
                     Witness : B; A : Q;
                     _________________________________
                     (pair Witness A) : (exists X : B P);) S))

(define exists-left
 {[sequent] --> [sequent]}
  S -> ((rule let P_fstA/X (subst [first A] X P)
             (first A) : B, (second A) : P_fstA/X >> R;
             ________________________________________
             A : (exists X : B P) >> R;) S))
```

Just to remind you of the proof assistant syntax, here is beginning of the first proof.

```
(4+) (proof-assistant _)

Input assumptions?  (y/n) y

1. [a : q]

Input assumptions?  (y/n) y

2. [p : [f a]]

Input assumptions?  (y/n) y

3. [c : [forall x : q [[f x] --> [g x]]]]

Input assumptions?  (y/n) n

Enter conclusion: [What? : [g a]]
==============================
Step 1        unsolved 1

?- [What? |:| [g a]]

1. [a |:| q]
2. [p |:| [f a]]
3. [c |:| [forall x |:| q [[f x] --> [g x]]]]
```

```
Tactic: answer
==============================
Step 2          unsolved 2

?- [What? |:| [g a]]

1. [a |:| q]
2. [p |:| [f a]]
3. [c |:| [forall x |:| q [[f x] --> [g x]]]]

Tactic: (forall-left a)
==============================
Step 3          unsolved 3

?- [a |:| q]

1. [c |:| [forall x |:| q [[f x] --> [g x]]]]
2. [a |:| q]
3. [p |:| [f a]]

Tactic:
```

Notice in step two I type (forall-left a) which is a partial application generating a function of type [sequent] --> [sequent] because forall-left is parameterised. You can probably complete the proof now.

end of post 5

We now pass on to proof by mathematical induction, which is really quite central to the whole process of extracting programs from proofs. Understanding this aspect requires a grasp of primitive recursion so I'll begin by a refresher course on primitive recursion.

Primitive Recursion

The n-ary functions that map from numbers to numbers are called number theoretic functions. One such class of number theoretic functions are the primitive recursive functions. This class in turn is a proper subset of the class of general recursive functions which occupy a significant place in the theory of computing. Their significance derives from the fact that the general recursive functions provide a model for computability wrt the natural numbers. We shall not be concerned with general recursive functions here, only primitive recursive functions.

The class of primitive recursive functions is specified recursively by isolating a set of base recursive functions and a higher order function which when applied to primitive recursive functions produces new primitive recursive functions. The primitive recursive functions are defined as follows.

1. A zero function is a primitive recursive function; it is definable in Qi as follows.

```
(define zero
  {A --> number}
   _ -> 0)
```

2. The successor function is primitive recursive

```
(define succ
 {number --> number}
  X -> (+ X 1))
```

3. An infinite set of projection functions is primitive recursive such that for any projection function fi:n of arity n (fi x1....xi...xn) = xi. A projection function is thus one that given a number of arguments always returns the ith one of them as its value.

Entering an infinite set of projection functions into Qi is obviously an impossibility, but we can gain the benefit of using projection functions by abstractions. Thus f1:3 can be represented by (/. X Y Z X).

4. Given two primitive recursive functions we can define a third by the operation of composition.

5. The higher order function prim can be used to define new primitive recursive functions. Its definition is

```
(define prim
 {number --> A --> (number --> A --> A) --> A}
 0 X _ -> X
 N X F -> (F (- N 1) (prim (- N 1) X F)))
```

Some Examples of Primitive Recursive Functions

Addition:

```
(define add
 {number --> number --> number}
  X Y -> (prim X Y (/. W Z (succ Z))))
```

Multiplication:

```
(define multiply
 {number --> number --> number}
 X Y -> (prim Y (zero X) (/. W Z (add X Z))))
```

Mathematical Induction in TT_0

Primitive recursion and TT_0 are connected together by TT_0's version of proof by mathematical induction which makes essential use of primitive recursion. We shall first state the rule in Qi II and then discuss its significance.

```
(define mathl-ind
 {[sequent] --> [sequent]}
  S -> ((rule let P0/X (subst 0 X P)
              let PsuccX/X (subst [succ X] X P)
              C : P0/X;
              F : (forall X : natnum (P --> PsuccX/X));
              _________________________________________
              (/. N (prim N C F)) : (forall X : natnum P);) S))
```

Ignoring the LHS, the rule is quite orthodox. Here we have just the RHS.

```
let P0/X (subst 0 X P)
let PsuccX/X (subst [succ X] X P)
P0/X;
(forall X : natnum (P --> PsuccX/X));
_________________________________
(forall X : natnum P);
```

It says that if we are given to prove a property P of all the natural numbers then we do it by

1. Proving the base case P0/X, where 0 replaces the bound variable X.
2. Proving the inductive case (forall X : natnum (P --> PsuccX/X)) where PsuccX/X is the result of replacing X by (succ X).

To understand the LHS we have to consider what an inductive proof delivers to us. Suppose there is an inductive proof that every natural number has the property F. Suppose we want to show that this gives us the means M of proving 5 has the property F.

This we can do by

a. Invoking our proof C of the base step 0.
b. Applying the proof F of the inductive step 5 times.

This thinking is reflected in the conclusion of our rule; (/. N (prim N C F)) is the means M just described.

```
(prim 5 C F) = (F 4 (F 3 (F 2 (F 1 (F 0 C)))))
```

Hence this rule gives is the basis for executing numerical computations from inductive proofs. Of course this is very important!

The last two rules are a mere addendum; we have to define the natural numbers as a type.

```
(define natnum-base
  {[sequent] --> [sequent]}
   S -> ((rule _________
               0 : natnum;) S))

(define natnum-induct
  {[sequent] --> [sequent]}
   S -> ((rule N : natnum;
               _______________
               (succ N) : natnum;) S))
```

In the next post we shall come to grips with the application of these ideas by actually synthesising a program.

____end of post 6

Now we come to the final part of the exposition.

Consider the proposition

Every natural number has an integer square root

Written in TT0 this appears as

(forall x : natnum (exists y : natnum (intsqrt y x)))

Now suppose we were to pose this as a problem

What? : (forall x : natnum (exists y : natnum (intsqrt y x)))

If we were to solve the problem, what would the solution for 'What?' look like?

By CTT theory it must be a function f that for any natural number n, delivers a proof of (exists y : natnum (intsqrt y n)). But what is a proof? of (exists y : natnum (intsqrt y n))? Again CTT tells use that this must be a pair <w, p> composed of a witness w that is the integer square root of n and a proof p of (intsqrt w n). But if this is true then the function f actually delivers or computes integer square roots. Hence from proving (forall x : natnum (exists y : natnum (intsqrt y x))) we can actually derive a program that computes them.

ergo programs = proofs

In CTT a program specification has the form

(forall x1 : t1 ... (forall xn : tn ... (exists y (R y x1...xn))))

where x1...xn are the inputs and y is the desired output. Proving this formula delivers the program.

An Actual Example

I'm going to choose an example fitted for a post. i.e not too long. We are going to synthesise the addition function.

We need two rules defining addition.

```
(define plus-base
  {[sequent] --> [sequent]}
   S -> ((rule _______________
                trivial : (plus X 0 X);) S))

(define plus-induct
  {[sequent] --> [sequent]}
   S -> ((rule A : (plus X Y Z);
              _______________________
              A : (plus X (succ Y) (succ Z));) S))
```

The first says that there is a trivial proof that x + 0 = x. Note we have to use a relational 'Prolog' style notation because we have not yet incorporated equality into TT_0.

The second says that a proof that x + (succ y) = (succ z) is also a proof that x + y = z.

We want to prove that for any two natural numbers there is a third which is their sum.

(forall x : natnum (forall y : natnum (exists z : natnum (plus x y z))))

The Proof

Here is the proof is all its monochrome glory.

```
(13+) (proof-assistant _)

Input assumptions?  (y/n) n

Enter conclusion: [What? : [forall x : natnum [forall y : natnum [exists z : natnum [plus x y z]]]]]
===============================
Step 1          unsolved 1

?- [What? |:|
 [forall x |:| natnum
  [forall y |:| natnum [exists z |:| natnum [plus x y z]]]]]

Tactic: answer
===============================
Step 2          unsolved 2

?- [What? |:|  [forall x |:| natnum   [forall y |:| natnum [exists z |:| natnum [plus x y z]]]]]

Tactic: all-right
===============================
Step 3          unsolved 2

?- [#:X22211 |:| [forall y |:| natnum [exists z |:| natnum [plus a659 y z]]]]

1. [a659 |:| natnum]

Tactic: mathl-ind
===============================
Step 4          unsolved 3

?- [#:X22220 |:| [exists z |:| natnum [plus a659 0 z]]]

1. [a659 |:| natnum]

Tactic: (some-right a659)
===============================
Step 5          unsolved 4

?- [a659 |:| natnum]

1. [a659 |:| natnum]

Tactic: hyp
```

```
==============================
Step 6          unsolved 3

?- [#:X22246 |:| [plus a659 0 a659]]

1. [a659 |:| natnum]

Tactic: plus-base
==============================
Step 7          unsolved 2

?- [#:X22223 |:|
 [forall y |:| natnum
  [[exists z |:| natnum [plus a659 y z]] -->
   [exists z |:| natnum [plus a659 [succ y] z]]]]]

1. [a659 |:| natnum]

Tactic: all-right
==============================
Step 8          unsolved 2

?- [#:X22274 |:|
 [[exists z |:| natnum [plus a659 a662 z]] -->
  [exists z |:| natnum [plus a659 [succ a662] z]]]]

1. [a662 |:| natnum]
2. [a659 |:| natnum]

Tactic: -->-right
==============================
Step 9          unsolved 2

?- [#:X22285 |:| [exists z |:| natnum [plus a659 [succ a662] z]]]

1. [#:X22282 |:| [exists z |:| natnum [plus a659 a662 z]]]
2. [a662 |:| natnum]
3. [a659 |:| natnum]

Tactic: some-left
==============================
Step 10         unsolved 2

?- [#:X22285 |:| [exists z |:| natnum [plus a659 [succ a662] z]]]

1. [[first #:X22301] |:| natnum]
2. [[second #:X22301] |:| [plus a659 a662 [first #:X22301]]]
3. [a662 |:| natnum]
4. [a659 |:| natnum]

Tactic: (some-right [succ [first #:X22301]])
```

```
==============================
Step 11        unsolved 3

?- [[succ [first #:X22301]] |:| natnum]

1. [[first #:X22301] |:| natnum]
2. [[second #:X22301] |:| [plus a659 a662 [first #:X22301]]]
3. [a662 |:| natnum]
4. [a659 |:| natnum]

Tactic: natnum-induct
==============================
Step 12        unsolved 3

?- [[first #:X22301] |:| natnum]

1. [[first #:X22301] |:| natnum]
2. [[second #:X22301] |:| [plus a659 a662 [first #:X22301]]]
3. [a662 |:| natnum]
4. [a659 |:| natnum]

Tactic: hyp
==============================
Step 13        unsolved 2

?- [#:X22506 |:| [plus a659 [succ a662] [succ [first #:X22301]]]]

1. [[first #:X22301] |:| natnum]
2. [[second #:X22301] |:| [plus a659 a662 [first #:X22301]]]
3. [a662 |:| natnum]
4. [a659 |:| natnum]

Tactic: plus-induct
==============================
Step 14        unsolved 2

?- [#:X22530 |:| [plus a659 a662 [first #:X22301]]]

1. [[first #:X22301] |:| natnum]
2. [[second #:X22301] |:| [plus a659 a662 [first #:X22301]]]
3. [a662 |:| natnum]
4. [a659 |:| natnum]

Tactic: hyp
```

```
==============================
Step 15        unsolved 1

?- [[[abs a659]
  [/. #:X22219
   [[[prim #:X22219] [[pair a659] trivial]]
    [[abs a662]
     [/. #:X22301 [[pair [succ [first #:X22301]]] [second
#:X22301]]]]]]]
 |:|
 [forall x |:| natnum
  [forall y |:| natnum [exists z |:| natnum [plus x y z]]]]]

1. [[[abs a659]
  [/. #:X22219
   [[[prim #:X22219] [[pair a659] trivial]]
    [[abs a662]
     [/. #:X22301 [[pair [succ [first #:X22301]]] [second
#:X22301]]]]]]]
 |:|
 [forall x |:| natnum
  [forall y |:| natnum [exists z |:| natnum [plus x y z]]]]]

Tactic: hyp
Real time: 266.5625 sec.
Run time: 0.1875 sec.
Space: 1083768 Bytes
GC: 2, GC time: 0.0 sec.
proved : symbol
```

The expression

```
[[[abs a659]
  [/. #:X22219
   [[[prim #:X22219] [[pair a659] trivial]]
    [[abs a662]
     [/. #:X22301 [[pair [succ [first #:X22301]]] [second #:X22301]]]]]]]
```

is what we actually want. Notice that this is annotated with 'abs' - an abstraction forming operator that we can define thus

```
(define abs
  [[abs X] Y] -> (let Z (newvar V)
                    (abs [/. Z (subst Z X Y)]))
  [X | Y] -> (map abs [X | Y])
  X -> X)
```

Lets put the program in a global

```
(set *rawprog* [[abs a659]
  [/. #:X22219
   [[[prim #:X22219] [[pair a659] trivial]]
    [[abs a662]
     [/. #:X22301 [[pair [succ [first #:X22301]]] [second #:X22301]]]]]]])
```

The raw program is a list representation of an executable. Lets get the real thing by applying 'abs' and pretty-printing the result in round brackets. We evaluate

```
(PPRINT (abs (value *rawprog*)))
```

and get

```
(/. V673
 (/. #:X22219
  (((prim #:X22219) ((pair V673) trivial))
   (/. V674
    (/. #:X22301 ((pair (succ (first #:X22301))) (second #:X22301)))))))
```

Qi doesn't like the #: so we dump them

```
(/. V673
 (/. X22219
  (((prim X22219) ((pair V673) trivial))
   (/. V674
    (/. X22301 ((pair (succ (first X22301))) (second X22301)))))))
```

We now have to define pair, first and second.

```
(define pair
  {A --> B --> (A * B)}
  X Y -> (@p X Y))

(define first
  {(A * B) --> A}
  X -> (fst X))

(define second
  {(A * B) --> B}
  X -> (snd X))
```

And finally we can test our addition program. Notice that we bind our synthesised program to a local Add before applying it to the two numbers X and Y. We take the first element of the result. Why? Because the function delivers a proof of the existential which is a pair <w, p> where w is the witness. What we want is the witness.

```
(14+) (define add
  {number --> number --> number}
  X Y -> (let Add (/. V673 (/. X22219
                              (((prim X22219) ((pair V673) trivial))
                           (/. V674
                              (/. X22301 ((pair (succ (first X22301))) (second X22301)))))))
             (first (Add X Y))))
add : (number --> (number --> number))
```

And then we test it.

```
(15+) (add 34 45)
79 : number
```

end of post 7

Conclusions

What can we conclude from this series of 7 posts? We can usefully divide our conclusions into

A. Remarks about the implementation.
B. Remarks about the ideas i.e. CTT itself.

Remarks about the Implementation

It should be clear that Qi II is an extremely powerful tool for investigating computational logic. We can prototype formal systems in Qi far more rapidly than in either Haskell, ML, O'Caml or Lisp. So if you're interested in any aspect of formalised reasoning, Qi II can give you a lot of leverage in allowing you to rapidly explore ideas.

If we were serious about this implementation we would do a number of things. If you're interested you can chase any of these things up and I'd be interested in any posts.

1. Implement the rest of TT_0 including rules for equality and list induction. Simon Thompson's book *'Type Theory and Functional Programming*' www.cs.kent.ac.uk/people/staff/sjt/TTFP gives those rules.
2. Give a proper syntax for the system. Saying anything is a wff is asking for trouble ;).
3. Devise a nicer proof assistant. What would nicer mean? Here's some ideas.
   a. Get away from entering stuff in list notation and more the (...) notation favoured in books.
   b. Move away from having to type in commands to point and click. This means using Qi/tk.
   c. Maybe too, being able to write formulae in non-ASCII form with true existential and universal quantifiers that are translated into an internal form.
4. Add a facility to store theorems in a library where they can be used.
5. Important; the use of unification in Qi II is itself not type secure. See http://www.lambdassociates.org/Book/page328.htm for a discussion.

In general there are two ways round this.

(i) Dispense with rule closures and code the system in standard Qi. This would result in a larger mass of code.
(ii) Keep the prototype and accompany it by a metaproof that the unification of any two wffs according to the syntax of the system must result in a wff. This is the way I would choose.

If those things were done then you would have a system like Coq/Elf/Lego/Nuprl, but based on Martin-Lof's work, with a code footprint that would be much smaller than any of them.

On Constructive Type Theory

CTT is very pretty. Is it practical? Our example suggests not; getting our heads around the synthesis of the addition function required far more effort than actually writing an addition function by hand. *But one example is misleading*. CTT might be very useful in other areas such as circuit specification say, where correctness might be specifiable quite elegantly and the cost of error is very high. So for safety critical work CTT might be quite useful.

The other observation is that our addition function is very inefficient. There are people working on this aspect. Actually there are some simple optimisations in our solution that would drastically improve performance. One is getting rid of the unnecessary currying in the solution which just generates redundant partial applications.

Mark Tarver