TYPE THEORY AND FUNCTIONAL PROGRAMMING

Thierry Coquand

Göteborg, November 1995

Part of this work is joint work with Thorsten Altenkirsh

Part comes from discussions with Dan Synek.

Proof Checking in Type Theory

LCF approach to Interactive Proof Checking: introduce an abstract data type of theorems.

In Type Theory, proofs are *explicit* objects, represented as functional expressions.

Proof Checking can be replaced by Type Checking.

Some Problems

The type-checker should be extremely reliable (small, and "clearly" correct). But Type Theory, in its original formulation, is a complex system; furthermore, a lot of issues are left unspecified:

1. how to represent terms?

2. how to implement substitutions?

3. how to deal with names? what is a constant? how to represent bound variables?

4. how to specify the type-checking algorithm?

In particular, the problem of $names$ has proved to be especially difficult, both theoretically and for the implementations. (This problem is not so visible in programming, because one seldom needs to compute open terms.)

Question: how to design and implement a proof system based on Type Theory?

A Method

We present a possible way to address these problems, based on the following principle:

Type Theory = Functional Programming + Dependent Types

This claim has to be refined in two ways:

1. We will be interested ultimately in a subpart of the programming language, the well-founded part (like in Strong Functional Programming, D. Turner), and the productive part,

2. We don't try to solve the problem of Type Inference, like in ML/Haskell; instead the main problem is *type-checking*: given a correct type A, is the expression e of type A.

Functional Programming

The connections between Type Theory and Functional Programming are deep and have played already an important rôle.
Curry-Howard's analogy: unification of logical connectives, like $\Rightarrow, \wedge$ and operators on types; modus ponens and functional application, ... This analogy goes quite far:

- *constructors* correspond to introduction rule,
- *canonical terms* correspond to proof in introduction form,
- *head reduction* corresponds to normalisation of proofs in natural deduction.

Functional Programming

We believe that this analogy goes further:

- *let expressions* correspond to local abreviations, definitions, and local lemmas,
- *case expressions* correspond to proofs by case analysis,
- uses of *recursion* correpond to inductive arguments.

Two main differences are the notions of *dependent types*, and *types as values*.

Claims:

- These two notions can be added, while preserving one of the main feature of (pure) functional programming: a "clean" denotational semantics,
- Type Theory can be seen as a terminating fragment of this extended programming language.

Expressions

The expressions are the one of the core of a functional programming language

```
M = x,y,z...            -- variables

    \ x -> M            -- abstraction

    M M                 -- application

    C M1 ... Mk         -- canonical expression

    let D in M          -- let(rec)

    case M {p -> M}     -- case expression

    (x:M)M              -- product

    data {C {x:M}}      -- labelled sum

    Type                -- the type of types
```

Expressions

For instance

```
N1 = data {True} : Type            -- unit type

N0 = data {} : Type                -- empty type
```

We can define types recursively

```
Nat = data {0; S (x:Nat)} : Type

list : Type -> Type

list = \ A -> data {Nil; Cons (x:A,xs:list A)}

eqNat : Nat -> Nat -> Type
```

Expressions

```
rel : Type -> Type

rel = \ A -> (x:A)(y:A)Type


ref : (A:Type)(R:rel A)Type

ref = \ A R -> (x:A)R x x


transclos : (A:Type)(R:rel A)rel A

transclos =
 \ A R x y ->
  data {C1 (p:R x y);
        C2 (p:R x y,q:transclos A R y z)}
```

Expressions

Remarks:

1. a constructor is simply a *label* for a labelled disjoint sum,
2. a data type is a recursively defined constant; its scope is defined like for other recursively defined constant,
3. we can use *let expressions* to give abbreviation for types; let expressions cannot be explained in term of abstraction and applications.

Denotational Semantics

We can adapt the Denotational Semantics of LML (Lennart Augustsson's thesis) to the present language.

The domain of values has to be extended with Type and Products.

The semantics will define a *value* $[\![M]\!]_\rho$ given a syntactical *expression* M and an *environment* ρ.

Notice that this definition is by *structural* induction on M; we don't have any problem of capture of variables here!

Correctness

Specification and implementation problem: how to be sure of the correctness of operations on terms?

Method: these operations have to be correct *semantically*.

In order to get both an easy proof of semantical correctness, and a concise implementation, we implement *symbolically* both the *expressions*, their *semantics* and the *evaluation function*.

Semantical Design

We follow the definition of models in Hindley-Seldin (Chapter 4): a model is a set D with an application $u\ v$ and an evaluation function $M\rho \in D$ for M expression and ρ environment, such that

- $((\lambda x\ M)\rho)\ a = M(\rho, x = a)$,
- $x\rho =$ lookup $x\ \rho$,
- $(M_1\ M_2)\rho = (M_1\rho)\ (M_2\ \rho)$,
- $(\lambda x\ M)\rho = (\lambda y\ N)\nu$ if for all elements $d \in D$, we have $M(\rho, x = d) = N(\nu, y = d)$, (Berry's rule).

This means that we have an explicit notion of $closures$.

All "syntactical" operations: reductions, conversions,...are done on closures, not on expressions.

Cf. P.L. Curien An abstract framework for environment machines TCS 82 (1991).

We $do\ not$ have explicit substitutions, in the sense of Abadi, Cardelli, Curien and Levy JFP 1 (1991): never push a substitution under an abstraction!

Semantical Design

In order to deal with conversion of "open terms", we use Berry's rule. To test $(\lambda x\ M)\rho = (\lambda y\ N)\nu$ it is "enough" to test $M(\rho, x = d) = N(\nu, y = d)$ for all $d \in D$.

In order to implement this, we introduce gensym values.

Thus, our implementation is not so surprising: it is like renaming systematically the variables, with a "suitable book keeping" mechanism.
But the way this is done, via closure and never pushing a substitution under an abstraction, is $crucial$ if one wants a simple correctness proof.

Applications

The meaning of $\lambda x \ M : (y : A)B$ should be that, for any $value\ v$ of type $[\![A]\!]$ the value $[\![M]\!]_{x=v}$ is of type $[\![B]\!]_{y=v}$.

This solves the problem of α-conversion found by R. Pollack in the substitution calculus!

One main syntactical difficulty in proofs is the substitution lemma

$$[\![M[x/N]]\!]_\rho = [\![M]\!]_{\rho, x=[\![N]\!]\rho}$$

Cf. Stoy, Denotational Semantics,
A. Stoughton, Substitution Revisited, TCS 59 (1988)

This problem disappears here, because we have explicit closures!

Finally, we never have to program any substitution functions! All the expressions we manipulate are $subexpressions$ of the given expressions.

Literature

Textbooks on first-order logic: 50% takes a structural definition (for instance, Enderton, Bell, Mendelson), and 50% a definition by induction on the length of the formula, with introduction of one constant for any element of the model (for instance Shoenfield, van Dalen).

In computer science, two books are extremely careful about the treatment of variables

- Stoy's book on Denotational Semantics,
- M. Hennessy Algebraic Theory of Processes.

Both proves the "substitution lemma", Stoy with Curry's definition of substitution, and Hennessy using Stoughton's definition.

For programming languages, usually, one does not need to define syntactical substitution. For instance

```
(\ x -> M) N = let x = N in M
```

is proved by use of denotional semantics.

Extensions of Type Theory

New constructs of the language have to be “justified” by giving their semantics.

For instance, let expressions, such as

```
let t = Bool : Type
in id t True
```

have a canonical denotational semantics.

By adding let expressions to Type Theory, we give a conceptual explanation of the notion of *constants* and toplevel environment.

More generally, we can follow the *design principles*, from denotational semantics, to Type Theory (cf. B. Tennent, D.Schmidt).

Extensions of Type Theory

Semantically, an environment can be thought of as a list of triples *identifiers*, *values* and (type) *values*.

We get a notion of *theories*; we have to extend the recursive definition of the domain of values.

By design, we have also a notion of parametrized theory!

This notion is inspired by the theory mechanism of PVS (SRI, Shankar, Lincoln, ...)

We adopt the dot notation for accessing the components of a theory.

An Example of a Theory

A new type, called Theory, is introduced.
An example: partial equivalence relation
(symetric and transitive).

```
theory_per
  (A:Set,R:A->A->Set,h1:sym A R,h2:trans A R) =

 theory

  {lem1 (x:A,y:A,p:R x y) = ... : R x x;

   lem2 (x:A,y:A,p:R x y) = ... : R y y;

   lem3 (x:A,y:A,z:A,p:R x z,q:R y z) = ... : R x y;

   ...

  }
 : Theory
```

At any point, we can form

```
let th0 = theory_per A0 R0 s0 t0
in  th0.lem2
    : (x:A0,y:A0,p:R0 x y)R0 y y
```

What is structural induction?

When is a recursive definition allowed in type theory?

The recursive calls should be done on "syntactical subparts" of the "formal arguments".

I think we can test this via a suitable notion of "symbolic evaluation".

Computation of normal form can be seen as an effective refined denotational semantics. This can be used for strictness analysis.

Conclusion

Type Theory = Functional Programming + Dependent Types

Benefits for Proof-Checking in Type Theory; Type Theory should have a simple denotational semantics, and should follow the design principles required for functional programming languages:

- a short, simple core part,
- explicit proof objects, and elegant notations for proofs,
- a method to ensure correctness of handling of names,
- an "heuristic" constraint for the implementation and possible extensions.

Benefits for Functional Programming:

We presented a programming language with dependent types, with

- a clear denotational semantics,
- with types as values, data types as first-class objects,
- with a notion of modularisation inspired from PVS.