Structured Type Theory

Chalmers University of Technology and
University of Göteborg

Catarina Coquand and Thierry Coquand

Introduction

We present our implementation, AGDA, of type theory. We limit ourselves in this presentation to a rather primitive form of type theory (dependent product with a simple notion of sorts) extended with structure facilities we find in most programming language: let-expressions (local definition) and a package mechanism. We call this language *Structured Type Theory*.

The first part describes the syntax of the language.

The second part contains a detailed description of a core language, which is used to implement Structured Type Theory. We present first an untyped language. The notion of meta-variables can be explained at this level. We give next a realisability semantics, and type-checking rules that are proven correct with respect to this semantics.

The third part explains how to interpret Structured Type Theory in this core language.

The main contributions are:

- use of explicit substitution to *simplify* and make *fully precise* the meta-theory of our system[1] and to allow the addition of meta-variables;
- a simplification of the constraint solving mechanism with respect to the work of [9], that has been actually tested in large examples in our implementation [3, 11],
- use of the notion of parametrised constants to a *package* mechanism, that has also been tested in our implementation, and brings some module structure without complicating the meta-theory of the system.[2]

The *type-checking* problem is: given a correct type A is the term e a correct term of type A? The *type-inference* (or type-synthesis) problem is: given a term e, is it correct and if so, what is its type? The main difference between our approach and other works [10, 13] is that we concentrate on the problem of type-checking and not of type inference. Another property that we do not require is uniqueness of types. It has been our experience that for application of type theory in proof checking, decidability of type-checking, even for a restricted class of terms, is enough.

Our presentation covers only a part of what is actually implemented in AGDA. We have not covered record types (structure and signature), nor data types and case expression.

1 Structured Type Theory

We extend type theory (with product and a hierarchy of universes) with let-expressions and packages.

[1] We hope that it would be possible to directly represent our definitions in a functional language such as Haskell.

[2] This should be compared to the approach in [5]: in this approach it is not possible to analyse reductions in an untyped way. Our approach is less powerful but has a relatively simple realisability interpretation.

1.1 Syntax

sort	::=	#n	
expr	::=	varid	
	|	`?`	Meta variables or holes
	|	`let` sdefs `in` expr	
	|	expr.varid	Selection
	|	\ varid `->` expr	λ-expression
	|	expr expr	Application
	|	sort	
	|	(varids `::` expr) `->` expr	Function type
sdef	::=	varid params `::` expr `=` expr	
def	::=	sdef	
	|	`postulate` varid `::` expr	
	|	`package` varid params `where` defs	
	|	`package` varid params `=` expr	
	|	`open` expr `use` opens	
sdefs	::=	{ sdef ; ... ; sdef }	
params	::=	(varids `::` expr) ... (varids `::` expr)	
open	::=	varid `::` expr `=` varid	
opens	::=	open , ... , open	
varids	::=	varid `,` ... `,` varid	
varid	::=	*string*	

We also have some syntactic sugar:

- `Set` is an abbreviation of #0 and `Type` of #1.
- *varid* `::` *expr* ; *varid* = *expr* is the same as *varid* :: *expr* = *expr*.
- There is also a infix notation, similar to Haskell, that will not be presented here.
- Instead of writing {..; ... ; ..} one can use the layout rule as in Haskell.
- We write *expr* `->` *expr* for the non-dependent function type
- The *open* construction `i :: A` is the same as `i :: A = i`.
- The definition `type x = A` is equivalent to `x :: Set = A`.
- The expression `open` *expr* `use` *opens* `in` *expr* is equivalent to `let open` *expr* `use` *opens* `in` *expr*

1.2 Informal Description

Most constructions are standard constructions of type theory or Automath like language [2]. We add the possibility of leaving some places yet incomplete with the "?" symbol. Each occurrence of "?" has to be interpreted as a different meta-variables. We add also the possibility of introducing postulates. The addition of let-expressions (local definition) is standard in most functional programming languages.

The notion of *package* allows to group together some definitions that share a common set of parameters. One can then instantiate these parameters and give a name to the instantiated package. One can also use any defined constant in a package, instantiating the parameters and making reference to the constant via the dot notation.

2 Core Language for Structured Type Theory

We present first an untyped language with reduction rules. We can use this language to give a realisability interpretation of type theory.

2.1 Expressions

We have a family of sets $T(X)$ indexed by a finite set X. Intuitively X represents the set of variables that has been declared at this stage. If $e \in T(x, y, z)$ an usual notation in logic is to write $e(x, y, z)$ to indicate that *at most* x, y, z may occur free in the expression e. This notation gets some meaning in the syntax we present.

We use the notation X, x for $X \cup \{x\}$, (and this supposes that x is not in X) and we have used the notation x, y, z for $\{x, y, z\}$.

The elements of $T(X)$ are now

- variables $x \in T(X)$ for $x \in X$,
- applications $c\ a \in T(X)$ for $c, a \in T(X)$; here we follow Schoenfinkel's notation of combinatory logic,
- abstraction $\lambda x\ e \in T(X)$ if x is not in X and $e \in T(X, x)$,
- instantiation $e\rho \in T(X)$ if $e \in T(Y)$ and ρ is a mapping $Y \to T(X)$.

Notice that, in general, $T(X)$ is not a subset of $T(Y)$ if $X \subseteq Y$. For instance we have $\lambda y\ x \in T(x)$ but $\lambda y\ x$ is *not* in $T(x, y)$. If $e \in T(x_1, \ldots, x_n)$ we may write $e(a_1, \ldots, a_n)$ instead of $e(x_1 = a_1, \ldots, x_n = a_n)$. Thus $e(x_1, \ldots, x_n)$ will mean $e(x_1 = x_1, \ldots, x_n = x_n)$.

We define $V(X) \subseteq T(X)$ the set of *values* by the clauses

- variables $x \in V(X)$ for $x \in X$,
- applications $c\ a \in V(X)$ for $c, a \in V(X)$;
- instantiation $e\rho \in V(X)$ if $e \in T(Y)$ and ρ is a mapping $Y \to V(X)$.

Notice that $V(X) \subseteq V(Y)$ if $X \subseteq Y$.

2.2 Constants

We may add *constants* that have a *name* c, a *body* e and a finite set of parameters X. Intuitively we have $c = e \in T(X)$. For instance we may introduce

$f = \lambda y\ x \in T(x)$.

We make a distinction between y which is a *variable* used for the argument of the function f and x which is a *parameter* of the definition of f. If $a \in T(X)$ we can consider $f(x = a) \in T(X)$ and we have for any $b \in T(X)$

$$f(x = a)\ b = (\lambda y\ x)(x = a)\ b = x(x = a, y = b) = a.$$

Like in Automath [2], we also introduce *primitive notions*. They are "postulated" constants that have only a name and a finite set of parameters. The notation is $c = \mathsf{PN} \in T(X)$.

2.3 Weak Conversion

The rules of *weak conversion* between values are

- $x\rho = \rho(x)$,
- $(a_1\ a_2)\rho = a_1\rho\ (a_2\rho)$,
- $(\lambda x\ e)\rho\ a = e(\rho, x = a)$,
- $(e(x_1 = a_1, \ldots, x_n = a_n))\rho = e(x_1 = a_1\rho, \ldots, x_n = a_n\rho)$,
- $c\rho = e\rho$ if c is a constant of body e.

Each of these equalities can be seen as a rewrite rules, and we have thus a rewriting system $\rightarrow$ at each level $T(X)$. We have then the following result, which is a generalisation of [6].

Proposition: The rewriting system $\rightarrow$ is confluent.

An expression that cannot be reduced is of the form $x\ a_1 \ldots\ a_n$ where each a_i cannot be reduced or $(\lambda x\ e)\rho$ where each expression in ρ cannot be reduced.

From now on, the values are considered up to *weak-conversion*.

Proposition: $e = e(x_1, \ldots, x_n)$ if $e \in V(x_1, \ldots, x_n)$.

2.4 Strong Conversion

We define another conversion relation on values which corresponds to equality of possibly infinite Böhm tree, up to η-conversion. This will be the greatest family of symmetric relations $\equiv_X$ on $V(X)$ such that if $v \equiv_X w$ and $v = (\lambda x\ e)\rho$ then we have $e(\rho, x = z) \equiv_{X,z} w\ z$ for z not in X and if $v = x\ v_1\ \ldots\ v_n$ then $w = x\ w_1\ \ldots\ w_n$ and $v_i \equiv_X w_i$.

Theorem: $\equiv_X$ is an equivalence relation on each $V(X)$ and it is a congruence: if $v_1 \equiv_X w_1$ and $v_2 \equiv_X w_2$ then $v_1\ v_2 \equiv_X w_1\ w_2$.

We can think of the equivalence class of a value as a possibly infinite Böhm tree. We say that a value is *normalisable* if its associated Böhm tree is finite. This can be described by an inductive definition: a value $x\ u_1\ \ldots\ u_n$ is normalisable if each u_i are normalisable, a value $(\lambda x\ e)\rho \in V(X)$ is normalisable if $e(\rho, x = z) \in V(X, z)$ is normalisable, and a value $c\rho\ u_1\ \ldots\ u_n$ where c is a primitive notion, is normalisable if each $\rho(x)$ is normalisable and each u_i is normalisable. A *neutral* value is a normalisable value of the form $c\rho\ u_1\ \ldots\ u_n$, where c is a primitive notion, or of the form $x\ u_1\ \ldots\ u_n$.

2.5 Metavariables and Constraints

We add metavariables that are like new constants $?f$ declared at one stage $?f \in T(X)$. A metavariable is intuitively like a constant that has not yet a definition. A *constraint* is an equation of the form $v_1 = v_2$ with $v_1, v_2 \in V(X)$. We can describe the simplifications of constraints at this untyped level.

Let us consider a constraint $?f\rho = x\ a_1 \ldots\ a_n$ where ρ is a renaming. We test first if x can be written $\rho(x')$ for some (unique) x'. If not, the constraint has no solution. If $x = \rho(x')$, we introduce *new* metavariables $?f_i \in T(X)$ and we define $?f = x'\ ?f_1 \ldots\ ?f_n$ and we add the constraints $?f_i\rho = a_i$.

We do a similar operation for a constraint $?f\rho = c(x_1 = a_1, \ldots, x_n = a_n)\ b_1\ \ldots\ b_m$ where c is a primitive notion and ρ is a renaming.

If $?f\rho = (\lambda x\ e)\nu \in V(X)$ where ρ is a renaming from Y to X, we choose a variable z not in Y and we introduce a new metavariable $?g \in T(Y, z)$; we define $?f = \lambda z\ ?g$ and we add the constraint $?g(\rho, z = y) = e(\rho, x = y)$ for some y not in X. Notice that $\rho, z = y$ is still a renaming.

Here are two examples, taken from [7], where we need renaming maps. We assume that $?f,\ ?g \in T()$ and we have the constraint

$$?f()\ z\ y = z\ (?g()\ y\ x) \in V(x, y, z)$$

We write then $?f = \lambda x \lambda y\ ?f_1$ and $?g = \lambda x \lambda y\ ?g_1$ with $?f_1,\ ?g_1 \in T(x, y)$ and we have the constraint

$$?f_1(x = z, y = y) = z\ ?g_1(x = y, y = x) \in V(x, y, z).$$

We then have, since $z = x(x = z, y = y)$ that $?f_1 = x\ ?f_2 \in T(x, y)$ and the remaining constraint

$$?f_2(x = z, y = y) = ?g_1(x = y, y = x) \in V(x, y, z).$$

(It is possible to see from this constraint that $?f_2 \in T(y)$; indeed it follows from the constraint that $?f_2(x = z, y = y)$ is in $V(z, y) \cap V(y, x) = V(y)$.)

Another example will be

$$?f()\ z\ y = x\ (?g()\ y\ x) \in V(x, y, z)$$

We write then $?f = \lambda x \lambda y\ ?f_1$ and $?g = \lambda x \lambda y\ ?g_1$ with $?f_1,\ ?g_1 \in T(x, y)$ and we have the constraint

$$?f_1(x = z, y = y) = x\ ?g_1(x = y, y = x) \in V(x, y, z).$$

Since x is not in the image of $(x = z, y = y)$ this last constraint has no solution.

3 A Realisability Model

We add products $[x : A]B \in T(X)$ if x not in X and $A \in T(X)$, $B \in T(X, x)$ and sorts $*_k \in T(X)$ for $k = 0, 1, \ldots$. The new weak conversion rules are

$$*_k \rho = *_k$$

and the new strong conversion rule is

$$\frac{A_1\rho_1 \equiv_X A_2\rho_2 \quad B_1(\rho_1, x_1 = z) \equiv_{X,z} B_2(\rho_2, x_2 = z)}{([x_1 : A_1]B_1)\rho_1 \equiv_X ([x_2 : A_2]B_2)\rho_2}$$

The results of confluence for weak reduction, and that strong conversion forms a congruence stay valid with this extension.

We can now follow [4] to describe a realisability model of type theory. The type-checking rules that we describe later are correct w.r.t. this model. To simplify, we leave implicit the (variable) finite index set of free variables.

We define, by induction on k, a set T_k of *types* (intuitively the types in the universe $*_k$) in such a way that $\mathrm{T}_0 \subseteq \mathrm{T}_1 \subseteq \ldots$ and simultaneously, we define for $A \in \mathrm{T}_k$ what is the set E_A of values of type A. This definition will be such that it is clear that E_A depends only on A and not of k such that $A \in \mathrm{T}_k$. We have $*_l \in \mathrm{T}_k$ if $l < k$ and $E_{*_l} = \mathrm{T}_l$ which is known by induction on k. Furthermore $([x : A]B)\rho \in \mathrm{T}_k$ if $A\rho \in \mathrm{T}_k$ and $B(\rho, x = u) \in \mathrm{T}_k$ whenever $u \in E_{A\rho}$. In that case $E_{([x:A]B)\rho}$ is the set of values w such that if $u \in E_{A\rho}$ then $w\ u \in E_{B(\rho,x=u)}$. Finally, any neutral value A is in T_k and E_A is the set of all neutral values.

A value is a *value type* iff it is in the union of the sets T_k.

A *semantical context* is a sequence $x_1 : A_1, \ldots, x_n : A_n$ with $A_i \in V(x_1, \ldots, x_{i-1})$ such that A_1 is a value type and $A_2(u_1) = A_2(x_1 = u_1)$ is a value type if $u_1 \in E_{A_1}$, $\ldots$, $A_n(u_1, \ldots, u_{n-1})$ is a value type if $u_1 \in E_{A_1}, \ldots, u_{n-1} \in E_{A_{n-1}(u_1,\ldots,u_{n-2})}$.

If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a context and $\rho = (x_1 = u_1, \ldots, x_n = u_n)$ we say that ρ *fits* Γ iff $u_1 \in E_{A_1}, \ldots, u_n \in E_{A_n(u_1,\ldots,u_{n-1})}$.

As in [4] we can show

Theorem: If A is a value type then A is normalisable, furthermore any element of E_A is normalisable. Furthermore, if $A \equiv B$ then A is a value type iff B is a value type and then $E_A = E_B$; if A is a value type, if $u \equiv v$ then $u \in E_A$ iff $v \in E_A$.

4 Typing Rules

A *context* $\Gamma, \Gamma_1, \ldots$ is a list of declarations $x_1 : A_1, \ldots, x_n : A_n$ with $x_i \neq x_j$ if $i \neq j$ and $A_i \in V(x_1, \ldots, x_{i-1})$. If Γ is a context $x_1 : A_1, \ldots, x_n : A_n$ we write $F(\Gamma)$ for the set $x_1 \ldots, x_n$ of variables of Γ. We may write $T(\Gamma)$, $V(\Gamma)$ instead of $T(X)$, $V(X)$ where $X = F(\Gamma)$. We write ρ_Γ the instantiation $x_1 = x_1, \ldots, x_n = x_n$.

We use Σ, Σ' to denote a list of constant declarations of the form $c : A = e\ \Gamma$ with $A, e \in T(\Gamma)$ that is

$$\Sigma = () \ \mid\ \Sigma, c : A = e\ \Gamma \ \mid\ \Sigma, c : A = \mathsf{PN}\ \Gamma$$

A typing judgement has the form $\Gamma \vdash_\Sigma e : A$ with $e \in T(\Gamma)$ and $A \in V(\Gamma)$ or the form $\Gamma \vdash A\ \mathsf{type}$ with $A \in T(\Gamma)$. We can leave Σ implicit. The type-checking rules for typing judgements are first

$$\frac{}{\Gamma \vdash *_k\ \mathsf{type}}$$

$$\frac{\Gamma \vdash A : *_k}{\Gamma \vdash A\ \mathsf{type}}$$

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x : A\rho_\Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash [x : A]B\ \mathsf{type}}$$

for the types, and then

$$\frac{}{\Gamma \vdash *_k : *_{k+1}}$$

$$\frac{\Gamma \vdash A : *_k}{\Gamma \vdash A : *_{k+1}}$$

$$\frac{x : A \in \Gamma \qquad A \equiv_\Gamma C}{\Gamma \vdash x : C}$$

$$\frac{\Gamma \vdash e_1 : ([x : A]B)\rho \quad \Gamma \vdash e_2 : A\rho \quad B(\rho, x = e_2\rho_\Gamma) \equiv_\Gamma C}{\Gamma \vdash e_1\ e_2 : C}$$

$$\frac{\Gamma, x : B\rho \vdash e : C(\rho, y = x)}{\Gamma \vdash \lambda x\ e : ([y : B]C)\rho}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A\rho_\Gamma \vdash B : s}{\Gamma \vdash [x : A]B : s}$$

$$\frac{c : A = e \ \in \Sigma \qquad \rho : \Gamma_1 \to \Gamma}{\Gamma_1 \vdash c\rho : A\bar{\rho}}$$

with

$$\frac{}{() : \Gamma_1 \to ()}$$

$$\frac{\rho : \Gamma_1 \to \Gamma \qquad \Gamma_1 \vdash a : A\bar{\rho}}{\rho, x = a : \Gamma_1 \to \Gamma, x : A}$$

where $\bar{\rho}$ denotes $x_1 = a_1\rho_{\Gamma_1}, \ldots, x_n = a_n\rho_{\Gamma_1}$ if $\rho = (x_1 = a_1, \ldots, x_n = a_n) : \Gamma_1 \to \Gamma$.
A list of definitions is defined by the following grammar

$$ds = () \ \mid \ c : A = e\ \Gamma, ds \ \mid \ c : A = \mathsf{PN}\ \Gamma, ds$$

We can then define when a list of definitions is correct

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash_\Sigma e : A\rho_\Gamma \qquad \vdash_{\Sigma, c:A=e\ \Gamma}\ ds}{\vdash_\Sigma c : A = e\ \Gamma, ds}$$

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \vdash_{\Sigma, c:A=\mathsf{PN}\ \Gamma}\ ds}{\vdash_\Sigma c : A = \mathsf{PN}\ \Gamma, ds}$$

We can now state the correctness of these typing rules with respect to our realisability semantics.

Theorem: If Γ is a semantical context, ρ fits Γ, $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma \vdash e : A\rho_\Gamma$ then $A\rho = (A\rho_\Gamma)\rho$ is a value type and $e\rho \in E_{A\rho}$.

We can see these type-checking rules as basic derivations between *judgements* all of the form

$$\frac{J_1, \ldots, J_m}{J}$$

where a judgement J is either a typing judgement $\Gamma \vdash_S a : A$ or an equality judgement $u_1 \equiv_X u_2$ between two values $u_1, u_2 \in V(X)$ and $m \leq 3$. We can define $J_1, \ldots, J_n \Longrightarrow J$ to mean that there is a derivation of J using such rules from the assumption $J_1, \ldots, J_m$. If we limit ourselves to term e of the following form

$$e \quad = \quad \lambda x\ e \mid [x : e_1]e_2 \mid s \mid x\ e_1\ \ldots\ e_n \mid c\rho\ e_1\ \ldots\ e_n$$

it should be clear that applications of the rules from a typing judgement $J = \Gamma \vdash e : A$ will either fail or produce a list of equality judgements $J_1, \ldots, J_n$ such that $J_1, \ldots, J_n \Longrightarrow J$.

Thus the type-checking rules can be interpreted by a deterministic algorithm $\tau(J) = J_1, \ldots, J_n$ that given a typing judgement J produces a list $J_1, \ldots, J_n$ of typing constraints or equality constraint.

Corollary: The judgements $\vdash e\ \mathsf{type}$ and $\vdash e : A$ for A value type are decidable.

4.1 Meta-variables

We may have meta-variables in the judgement. If we add now to our syntax for terms

$$e \quad = \quad ?_k \mid \lambda x\ e \mid [x : e_1]e_2 \mid s \mid x\ e_1\ \ldots\ e_n \mid c\nu\ e_1\ \ldots\ e_n$$

then applications of the rules from a typing judgement $J = \Gamma \vdash e : A$ will either fail or produce a list of judgements $J_1, \ldots, J_n$ that are either *equality constraints* or *typing constraints* of the form $\Gamma_k \vdash ?_k : A_k$ such that $J_1, \ldots, J_n \Longrightarrow J$. The constraints are displayed to the user, but only as a help to find the value of some metavariables. As explained above, some simplifications of constraints may be done automatically.[3]

The *refinement* operation is then the following: if we have a typing constraint of the form $\Gamma_k \vdash ?_k : A_k$ the refinement $?_k = e$ will consist in substituting everywhere $?_k$ by e[4] and replacing $J' = \Gamma_k \vdash ?_k : A_k$ by $\tau(\Gamma_k \vdash e : A_k)$. If $?_k$ has only one typing constraint the other typing constraints stay as typing constraints, and the equational constraints $u \equiv_X v$ becomes $u(?_k/e) \equiv_X v(?_k/e)$.

The new idea with respect to the work of Lena Magnusson is of introducing systematically *new* metavariables: in a refinement $?_k = e$, the expression e should contain only new meta-variables [5]. The heuristic is to not introducing new sharing in the representation of terms. In this way, to any meta-variable correspond *exactly one* context and *exactly one* expected type, and the problem of checking dependency conditions between meta-variables does not arise.

5 Adding Let-expressions

There are now two alternatives for translating Structured Type Theory into the core language. Both have been implemented. One is to interpret all definitions, global or local, of Structured Type Theory, as definition of constants in the core language. The other is to add local let-expressions in the core language. In the second alternative, we add to the expressions

$$[x : A = e_1]e \in T(X)$$

with $A, e_1 \in T(X)$ and $e \in T(X, x)$. The new weak conversion rule is

$$([x : A = e_1]e)\rho = e(\rho, x = e_1\rho)$$

while the type-checking judgement have now the form $\Gamma, \rho, \Gamma_1 \vdash A\ \mathsf{type}$ and $\Gamma, \rho, \Gamma_1 \vdash e : A$ where Γ is the context of free variables, Γ_1 the context of bound variables $y_1 : B_1, \ldots, y_m : B_m$ with $B_j \in V(\Gamma)$ and ρ is the identity on the free variables and gives a value in $V(\Gamma)$ to each bound variables y_j. The new typing rule is

$$\frac{\Gamma, \rho, \Gamma_1 \vdash A_1\ \mathsf{type} \quad \Gamma, \rho, \Gamma_1 \vdash e_1 : A_1\rho \quad \Gamma, (\rho, x = e_1\rho), \Gamma_1[x : A_1\rho] \vdash e : A}{\Gamma, \rho, \Gamma_1 \vdash [x : A_1 = e_1]e : A}$$

and the other typing rules have to be changed accordingly. For instance the rule for typing product becomes

$$\frac{\Gamma, \rho, \Gamma_1 \vdash A\ \mathsf{type} \quad \Gamma[x : A\rho], (\rho, x = x), \Gamma_1 \vdash B\ \mathsf{type}}{\Gamma, \rho, \Gamma_1 \vdash [x : A]B\ \mathsf{type}}$$

6 Translation

The translation of Structured Type Theory to the core language is quite straight forward and we will only give an example.

[3] However, it may be that, using constants, we find a possible expression for a metavariable shorter than the one suggested by a constraint. Thus, in our implementation, the user may always choose if a constraint should be solved automatically or not.

[4] It is a simple substitution operation, since there is no possible captures, because of the use of explicit substitution.

[5] Syntactically, this is ensures by parsing only ?, which is interpreted as a fresh metavariable. It follows also from this that in all typing constraint $\Gamma_k \vdash ?_k : A_k$ the place-holder $?_k$ has no occurrence in Γ_k, A_k.

6.1 Example

The following fragment

```
postulate B :: Set
g (x::B) :: B = x
package M (A::Set)(x::A)(y::B) where
     c(z::A):: B = let f(x::B) :: B = x
                   in g x
     f :: B = c x
h (x,z::B) :: B = (M B x z).c (g ?)
```

is translated into in the first approach

$$
\begin{array}{l}
B \ : \ *_0 \ = \ \mathsf{PN} \ [] \\
g \ : \ [x : B()]B() \ = \ \lambda x \ x \ [] \\
f \ : \ [x1 : B()]B() \ = \ \lambda x1 \ x1 \ \ [A : *_0, x : A, y : B(), z : A] \\
M.c \ : \ [z : A]B() \ = \ \lambda z \ g() \ x \ \ [A : *_0, x : A, y : B()] \\
M.f \ : \ B() \ = \ M.c \ x \ \ [A : *_0, x : A, y : B()] \\
h \ : \ [x : B()][z : B()]B() = \lambda x \ \lambda z \ M.c(A = B(), x = x, y = z) \ (g() \ ?_1)
\end{array}
$$

In the second approach this will be translated into

$$
\begin{array}{l}
B \ : \ *_0 \ = \ \mathsf{PN} \ [] \\
g \ : \ [x : B()]B() \ = \ \lambda x \ x \ [] \\
M.c \ : \ [z : A]B() \ = \ \lambda z \ [f \ : \ [x1 : B()]B() \ = \ \lambda x1 \ x1]g() \ x \ \ [A : *_0, x : A, y : B()] \\
M.f \ : \ B() \ = \ M.c \ x \ \ [A : *_0, x : A, y : B()] \\
h \ : \ [x : B()][z : B()]B() = \lambda x \ \lambda z \ M.c(A = B(), x = x, y = z) \ (g() \ ?_1)
\end{array}
$$

7 Possible Extensions

We can add an abstraction mechanism. The syntax is

```
package M (A::Set) where
     c1 :: A1 = e1
     abstract c2 :: A2 = e2
     c3 :: A3 = e3
     abstract c4 :: A4 = e4
```

and the effect is that the definitions of $c2, c4$ are hidden when used from outside the package M, and $e2, e4$ can be changed without having to change $e1$ nor $e3$.

A first extension is to add sigma types, with pairs. This is what is needed to represent mathematical structures. In such a system, one can represent conveniently abstract algebra [11] and category theory [12].

A more delicate extension is the addition of inductive data types with recursively defined functions over these data types. One way is to add new reduction rules at the untyped level, and to represent data types and constructors as primitive notions.

References

[1] G. Barthe and M.H. Sörensen. Domain-Free Pure Type Systems. to appear in Journal of Functional Programming, 1999.

[2] N.G. de Bruijn. (1970) The mathematical language AUTOMATH, its usage, and some of its extensions. Symposium on automatic demonstration, Versailles 1968, LNM 125, pp. 29-61.

[3] J. Cederquist. A Pointfree approach to Constructive Analysis in Type Theory. PhD thesis, Chalmers University of Technology, 1997.

[4] C. Coquand. A realizability interpretation of Martin-Löf's type theory. In: 25 years of Type Theory, G. Sambin and J. Smith eds, Oxford University Press.

[5] J. Courant. MC: A Module Calculus for Pure Type Systems. Tech. Report, Orsay, nr. 1217.

[6] P.L. Curien. (1991) An abstract framework for environment machines. Theoret. Comput. Sci. 82 (1991), 389–402.

[7] G. Dowek, Th. Hardin, Cl. Kirchner, F. Pfenning. (1998) Unification via Explicit Substitutions: The Case of Higher-Order Pattern. INRIA report, 3591.

[8] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic.* Springer-Verlag, 1992.

[9] L. Magnusson. *The Implementation of ALF, a Proof Editor based on Martin-Löf monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology, 1995.

[10] C. Munoz. *Dependent Type Systems with Explicit Substitutions.* PhD thesis, INRIA, 1997.

[11] H. Persson. *Integrated Developments in Type Theory.* Ph. D. thesis, Chalmers University of Technology, 1999.

[12] A. Saibi. *Category Theory in Type Theory.* Ph. D. thesis, INRIA, 1999.

[13] M. Luther and M. Stecker. *A guided tour through TYPELAB.* Report, University of Ulm.