Course Notes in Typed Lambda Calculus

Thierry Coquand
Chalmers University

September 2008

Introduction

Since quite good books [10, 8] or review article [3, 14] are available on typed lambda calculi, these notes limit themselves to some historical remarks and some points that I consider delicate/important.

1 History of Lambda-Calculus and Types

Both have a common origin in logic. The first notion of types seem to come from Frege: he revealed the conceptual difference between objects and predicates and considered the hierarchy built on these notions. At the "bottom" level we have a collection of basic objects. Then we have properties of these objects. Then we may have properties over these properties. Typically the quantifiers (introduced by Frege) are of such a nature: to be universally true is a property of properties. In such a view, it makes no sense for instance to apply a quantifier to an object.

It is thus *not* possible to apply what is known as Russell's paradox in this logic. The predicate of predicates P that do not hold on themselves, that is $\neg P(P)$ hold, cannot simply be formed in this system! Instead the problem with Frege's system was that Frege supposed that for each predicate P there was associated an object $\bar{P}$ the *extension* of P. Furthermore, it was supposed that $\bar{P}_1 = \bar{P}_2$ implies that $P_1(x)$ and $P_2(x)$ are equivalent for any object x. We can now form the predicate $P_0(x)$ defined by

$$\exists Q\ [[\bar{Q} = x] \wedge \neg Q(x)]$$

and prove that $P_0(\bar{P_0})$ is equivalent to $\neg P_0(\bar{P_0})$, and hence deduce a contradiction.

It seems that the lambda-calculus notation originates from the representation of classes in Principia Mathematica of Russell and Whitehead. The class, or predicate, of terms x satisfying a property ϕ is written there $\hat{x}\phi$. Thus, for instance the class of predicates x that do not satisfy themselves, used in Russell's paradox is the class $R\ =\ \hat{x}\neg x(x)$. Frege had already a notation for functional abstraction. Church, in the 1930s, analysing Principia Mathematica, introduced the powerful λ notation, writing $\lambda x\ t$ instead of $\hat{x}\ t$, and explicited the crucial β conversion rule

$$(\lambda x\ t)\ u\ =\ [x/u]t.$$

Notice that $\lambda x\ \phi$ represents a predicate which is satisfied by the objects a such that $[x/a]\phi$ holds. Thus, the first purpose of λ-terms was clearly to represent *formulae* and *predicates.*

Church thought first it would be possible to avoid Russell's paradox without introducing types, but by staying within an intuitionistic logic that use only some limited form of the law of excluded middle [4]. This system was however shown to be inconsistent by his students Kleene and Rosser [13][1]. Church formulated then an elegant formulation of higher-order logic, using simply typed λ-calculus [5], which can be seen as a simplification of the type system used in Principia Mathematica, but also is in some sense a return to Frege.

2 Definition of Simply Typed Lambda Calculus

2.1 Notation

Lambda calculus introduces a direct notation for functions. In mathematics there is no direct notation for functions, but we have to refer to them by names. Thus we would say something like: let f be the map $x \longmapsto x^2 + 3$ and use f. Sometimes, one can even refer to f as the function $x^2 + 3$ but this may be confusing when several variables are involved. For instance $g : x \longmapsto x^2 + y$ where y is a parameter. The lambda notation allows to refer directly to these maps, respectively as $\lambda x\ x^2 + 3$ and $\lambda x\ x^2 + y$. (This notation is so convenient that one often wonders why it is not used in mathematics.)

2.2 Syntax

First we define types: we have a set of basic types, and if α, β are types then $\alpha{\rightarrow}\beta$ is a type. A convenient notation, when more than two types occur, is to associate to the right, so that we write $\alpha_1{\rightarrow}\alpha_2{\rightarrow}\alpha_3$ for the type $\alpha_1{\rightarrow}(\alpha_2{\rightarrow}\alpha_3)$. For each α, we suppose given a set V_α of variables of type α in such a way that V_α and V_β are disjoint if $\alpha \neq \beta$. Next, the set T_α of terms of type α is defined as follows:

- if $x \in V_\alpha$ then $x \in T_\alpha$,
- if $t \in T_\beta$ and $x \in V_\alpha$ then $\lambda x\ t \in T_{\alpha\rightarrow\beta}$,
- if $t \in T_{\alpha\rightarrow\beta}$ and $u \in T_\alpha$ then $t\ u \in T_\beta$.

We define next the notion of free variables of a term: x is free in t iff

- $x = t$ or
- $t = t_1\ t_2$ and x is free in t_1 or t_2, or
- $t = \lambda y\ u$ and $x \neq u$ and x is free in u.

Exercice: We can define like in untyped lambda calculus the notion of term in normal form. Show that a term in normal form can be written $\lambda x_1 \dots \lambda x_k\ x\ M_1 \dots M_l$ where we can have $k = 0$ or $l = 0$ and $M_1, \dots, M_l$ are in normal form.

Use this to enumerate the closed terms of the following types (ι is a ground type)

1. $\iota{\rightarrow}\iota$,
2. $\iota{\rightarrow}\iota{\rightarrow}\iota$,
3. $(\iota{\rightarrow}\iota){\rightarrow}\iota{\rightarrow}\iota$,
4. $\iota{\rightarrow}(\iota{\rightarrow}\iota){\rightarrow}\iota$,

[1]What is remarkable is that they obtained their paradox by a formalisation in Church's system of Richard's paradox about the "least integer definable in less than 100 words." (Poincare had stressed the importance of this paradox.)

5. $(\iota \to \iota) \to \iota$.

6. $((\iota \to \iota) \to \iota) \to \iota$.

2.3 Models and Conversion

We present here a model theoretic definition of conversion, that avoids subtle syntactical considerations (alpha-conversion, clashes of variables) that are rarely treated rigourously.

2.3.1 Henkin's Definition

The intended model is the set-theoretic model where we suppose given a set D_ι for each base type ι and where $D_{\alpha \to \beta}$ is the set of all functions from D_α to D_β. We define then typed environments that are finite list of pairs $x = a$ where $x \in V_\alpha$ and $a \in D_\alpha$; the variables appearing in ρ form a set called the *domain* of ρ and we write $x\rho$ the lookup of x in ρ for any x in the domain of ρ. By induction on t it is possible to give a meaning $t\rho$ for any environment ρ whose domains contain all the free variable of t. It is defined by the laws

1. $x(\rho, y = b) = x\rho$, if $x \neq y$,

2. $x(\rho, x = a) = a$,

3. $(\lambda x\ t)\rho = a \longmapsto t(\rho, x = a)$, $\qquad (*)$

4. $(t\ u)\rho = t\rho\ (u\rho)$.

Henkin generalised this model by starting from a collection D_α of sets indexed by the types, such that $D_{\alpha \to \beta}$ is only a *subset* of the set of all functions from D_α to D_β. The intuition is that the objects of the models should be only those functions which preserve some given structure. Now, in general given such a collection, it is not possible to define the meaning function $t\rho$ because in the clause $(*)$ it may not be the case that the function

$$a \longmapsto t(\rho, x = a)$$

belongs to $D_{\alpha \to \beta}$.

Henkin defines a *model* to be such a collection D_α such that this recursive definition of $t\rho$ is possible.

As Henkin noticed himself, this definition has some impredicative flavour.

2.3.2 Alternative Definition

A more algebraic definition of model is the following. We suppose given for each type α a set D_α with an application map $D_{\alpha \to \beta} \times D_\alpha \to D_\beta$ that will be written as concatenation. We define then typed environments that are finite list of pairs $x = a$ where $x \in V_\alpha$ and $a \in D_\alpha$. Finally, we suppose given for each term $M \in T_\alpha$ and each environment ρ containing all the free variable of M an element $M\rho \in D_\alpha$ such that the following equations are satisfied:

1. $x(\rho, y = b) = x\rho$, if $x \neq y$,

2. $x(\rho, x = a) = a$,

3. $(\lambda x\ t)\rho\ a = t(\rho, x = a)$,

4. $(t\ u)\rho = t\rho\ (u\rho)$.

These equations are quite natural. The first two equations simply says that $x\rho$ is the lookup of the value of x in the environment ρ while the third equation says that $(\lambda x\ t)\rho$ can be thought of as the function $a \longmapsto t(\rho, x = a)$, which is the intended semantics of lambda abstraction. Finally, the last equation interprets application as ordinary function application.

So far, we get only a notion of *weak* model, which is more general than Henkin's notion of model. In order to get back the same notion of models as Henkin's model, we have to suppose furthermore the following *extensionality condition* that $f = g \in D_{\alpha\to\beta}$ whenever $f\ a = g\ a$ for all $a \in D_\alpha$. If the model satisfies this extensionality condition, we can think of $D_{\alpha\to\beta}$ as a subset of $D_\alpha^{D_\beta}$. In this model, it can be checked that the law of *η-conversion* is valid: if x is not a free variable of t than for any environment ρ for t

$$(\lambda x\ t\ x)\rho = t\rho$$

because for any a we have

$$(\lambda x\ t\ x)\rho\ a = (t\ x)(\rho, x = a) = t\rho\ a.$$

If we want a model of β-conversion only, we have to replace the extensionality condition by the weaker so-called *Berry condition*: we have $(\lambda x\ u)\rho = (\lambda y\ v)\nu$ iff for any value a we have $u(\rho, x = a) = v(\nu, y = a)$.

2.3.3 Conversion

We can now define conversion at each types: two terms t and u in T_α are convertible iff for any model D we have $t\rho = u\rho \in D_\alpha$ for any environment giving a value for each free variables in t and in u.

For instance $t = (\lambda x\ x)y$ and y are convertible if x, y are of type α because for any model D and any environment ρ giving a value to x and y we have

$$t\rho = (\lambda x\ x)\rho\ (y\rho) = x(\rho, x = y\rho) = y\rho.$$

In the same way $f \in V_{\alpha\to\alpha}$ and $\lambda x\ f\ x$ are convertible because for any $a \in D_\alpha$

$$(\lambda x\ f\ x)\rho\ a = (f\ x)(\rho, x = a) = f(\rho, x = a)\ (x(\rho, x = a)) = f\rho\ a.$$

2.4 Examples of Models

2.4.1 Set Theoretic Model

We take for D_ι, where ι is a base type, an arbitrary set and we interpret $D_{\alpha\to\beta}$ as the set of *all* set theoretic functions from D_α to D_β. (We recall that, in set theory, a function is identified with a functional relation.) There is then now problem to define $t\rho$ by induction on t :

- if $t = x$ then $t\rho$ is the lookup of x in ρ,
- if $t = t_1\ t_2$ then we define inductively $t\rho$ as the value of the function $t_1\rho$ on the argument $t_2\rho$,
- if $t = \lambda x\ u$ then $t\rho$ is the function that associates to the value a the value $u(\rho, x = a)$ which is defined by induction hypothesis.

2.4.2 Domain Model

The previous model may be thought of as the intended model. However, in general, the semantics of a term is a functional preserving some structures. (We will see later that the definable functionals satisfy also remarkable uniformity property.) A typical example is the domain model. The set D_ι, for ι base types, are now partial order sets with a bottom element $\perp$ and sups of increasing chains. The set $D_{\alpha\rightarrow\beta}$ is the poset, for the pointwise ordering, of the set of all increasing maps $f : D_\alpha \rightarrow D_\beta$ preserving the sups of the increasing chains, that is $f\ (\vee x_n) = \vee(f\ x_n)$ for any chain $x_0 \leq x_1 \leq x_2 \ldots$

When trying to define the meaning of lambda terms in these domains, there is a difficulty in the abstraction case, for defining $(\lambda x\ u)\rho$, which is typical of the definition of models of typed lambda calculus. We should define it as the map $a \longmapsto u(\rho, x = a)$ where, for each a, we have defined inductively the value $u(\rho, x = a)$. The difficulty is that we don't know that this value depends *continuously* on a.

In order to solve this problem, we have to express and check inductively that the value of $t\rho$ depends continuously on ρ. Thus, we define inductively on $t \in T_\alpha$ a continuous function

$$(a_1, \ldots, a_k) \longmapsto t(x_1 = a_1, \ldots, x_k = a_k)$$

from $D_{\alpha_1} \times \ldots \times D_{\alpha_k}$ to D_α. This formulation incorporates in the induction both the construction of $t\rho$ and the fact that $t\rho$ depends continuously on ρ. The problem is reduced to the following lemma, whose proof is a direct consequence of the definitions.

Lemma: If $f : D_1 \times D_2 \rightarrow D$ is a continuous function then the function $a \longmapsto \lambda y\ f(a, y)$ from D_1 to $[D_2 \rightarrow D]$ is a continuous function.

One interest of this model is that there is a fixedpoint operator $Y_\alpha \in D_{(\alpha\rightarrow\alpha)\rightarrow\alpha}$ defined as $Y_\alpha\ f = \vee(f^n\ \perp)$. It can indeed been checked that the sequence $f^n\ \perp$ is increasing and that $Y_\alpha\ f$ depends continuously on f.

There is something quite remarkable about the cardinality of the domains D_α, which is indeed described by Dana Scott as his "first original discovery" in the theory of domains [17]. If we start from domains D_ι such that there exists a countable basis $B_\iota \subseteq D_\iota$, that is a subset such that any element can be written as a sup of an increasing chain of elements in B_ι, then it can be checked that any domain D_α, even at higher types, has a countable basis. It follows that the cardinality of each domain is at most the one of the continuum. Actually, a rigourous and satisfactory description of these domains is best done by an explicit construction of each basis, and this is achieved by the notion of *information system*, see [20].

2.4.3 Recursive Model

I refer here to [14]. Intuitively, this model keeps only the functions and functionals that are "computable". It is important to realise that the meaning of computable is quite subtle however. At type $\iota\rightarrow\iota$, it coincides with the notion of *recursive* function. At type $(\iota\rightarrow\iota)\rightarrow\iota$, it is not clear if to be "computable" at this type should imply that it is defined only for computable functions or on an arbitrary function.

2.4.4 Term Model

This is the free model. Its construction is quite similar is the construction of a polynomial algebra. An intuitive description is that we build a model in a "syntactical" way, built freely from an infinite set of parameters P_α at each type α.

We suppose given for each type α a countable infinite set P_α of parameters of type α and we define the set Val_α of values of type α by the clauses:

- each element of P_α is a value of type α,
- if $M \in T_\alpha$ and ρ a list of pairs $x = a$ where $a \in D_\tau$ and $x \in V_\tau$ containing all the free variables of M then $M\rho \in Val_\alpha$,
- if $c \in Val_{\alpha\rightarrow\beta}$ and $a \in Val_\alpha$ then $c\ a \in Val_\beta$.

The conversion relation $a = a' \in D_\alpha$ is the least equivalence congruence such that

1. $x(\rho, y = b) = x\rho$, if $x \neq y$,
2. $x(\rho, x = a) = a$,
3. $(\lambda x\ t)\rho\ a = t(\rho, x = a)$,
4. $(t\ u)\rho = t\rho\ (u\rho)$,
5. $c\rho = c'\rho'$ if $c\rho\ v = c'\rho'\ v$ where v is a parameter not in ρ, ρ'.

It should be clear that this defines a model. Furthermore, this model has the initial property that, for any other model (D_α), any system of maps $P_\alpha \rightarrow D_\alpha$ can be extended to a morphism $f_\alpha : Val_\alpha \rightarrow D_\alpha$ in a unique way. (To be a morphism means that $f\ (w\ u) = f\ w\ (f\ u)$ and that $f\ (t\rho) = t(f^*\ \rho)$ where $f^*\ \rho$ is defined by $x(f^*\ \rho) = f\ (x\rho)$.)

It is then clear that to check if two terms of the same type are convertible, it is enough to check that they have the same value in this particular model. Furthermore, it is enough to interpret the free variables of these terms by parameters. This is intuitively clear as shown in the following example. The terms $t_1 = (\lambda x\ y\ x)\ z$ and $t_2 = y\ z$ where $y \in V_{\alpha\rightarrow\alpha}$ and $z \in V_\alpha$ are convertible, since if $w \in P_{\alpha\rightarrow\alpha}$ and $v \in P_\alpha$ we have, for $\rho = (y = w, z = v)$

$$t_1\rho = (\lambda x\ y\ x)\rho\ (z\rho) = (\lambda x\ y\ x)\rho\ v = (y\ x)(\rho, x = v) = y(\rho, x = v)\ (x(\rho, x = v)) = w\ v = t_2\rho$$

This term model is a nice illustration of Skolem "paradox" since all types are interpreted by a countable set. In Henkin's paper [11], where such a construction is first defined, it is explicitely noted that the existence of a countable model is not clear a priori, given the apparent impredicativity or circularity of the notion of models (we have to give a set D_α for all types α). In contrast, both the set-theoretic and the domain models are determined by the choice of the domains at base types and built by induction on the type hierarchy.

2.4.5 Implementation Model

We extend the language with one *fixpoint* operator $Y : (\alpha\rightarrow\alpha)\rightarrow\alpha$ and we consider weak models that are models with the property that $Y\ u = u\ (Y\ u)$ for any $u \in D_{\alpha\rightarrow\alpha}$.

3 Logical Relations

Given a binary relation R_ι on each D_ι for ι base type, we extend this relation at higher types by defining $R_{\alpha\rightarrow\beta}(t, t')$ to be $R_\alpha(u, u') \Rightarrow R_\beta(t\ u, t'\ u')$. We say that two environments ρ and ρ' are related iff $x\rho$ is defined iff $x\rho'$ is defined and then $R_\alpha(x\rho, x\rho')$ for any variable x of type α. The following result is known as the *Fundamental Theorem of Logical Relations*:

Theorem: for any term t of type α we have $R_\alpha(t\rho, t\rho')$ if ρ and ρ' are related.

Proof: By a direct induction on the term t. If $t = x$ it is $R_\alpha(x\rho, x\rho')$ which holds because ρ and ρ' are related. If $t = t_1\ t_2$ then by induction we have $R_{\alpha_2 \to \alpha}(t_1\rho, t_1\rho')$ and $R_{\alpha_2}(t_2\rho, t_2\rho')$ and hence $R_\alpha(t\rho, t\rho')$. If t is $\lambda x.u$ with $\alpha = \beta \to \gamma$ then for any a, a' in V_β such that $R_\beta(a, a')$ holds we have, by induction

$$R_\gamma(u(\rho, x = a), y(\rho', x = a')$$

since $R((\rho, x = a)(y), (\rho', x = a')(y))$ for any free variable y of u. This implies $R_\alpha(t\rho, t\rho')$ as desired. $\square$

This theorem deserves its name, for, while its proof is rather simple, most of the results about lambda calculus can be seen as corollary of this result.

More generally, a simple modification of the definition gives the notion of logical relation between the elements of two different models. Another generalisation concerns the addition of constant; one simply then check that each constant has its interpretation related to itself.

3.1 Definability Problem

Given the strong uniformity condition of definable terms, one can ask if conversely, this condition is enough to ensure that a term is definable. A related question is the *definability problem*: is there an algorithm that computes when a given object $a \in D_\alpha$ built from given *finite* sets is definable? It was shown by Sieber [18] that the answer is positive if α is of order ≤ 2 (where the order of a ground type is 0 and the order of $\alpha \to \beta$ is the maximum of the order of β and the successor of the order of α.) A surprising result of Loader is that the answer is *not* decidable at order 3! An implementation of Sieber's algorithm is described in [18] and is a nice application of logical relation and of representation of inductive definitions in second-order logic.

Exercice: We can easily show that there exists no term $t : B \to B$ which may use the constant $0, 1 : B$ such that $t\ 0 = 1$ and $t\ 1 = 0$. Indeed if such a term exists, we would have by the fundamental theorem of logical relation

$$(R)[R(0,0) \to R(1,1) \to R(0,1) \to R(t\ 0, t\ 1)]$$

that is

$$(R)[R(0,0) \to R(1,1) \to R(0,1) \to R(1,0)].$$

But this formula is directly seen to be not valid.

Exercice: We describe Stoughton's algorithm on a simple concrete instance. Suppose that we want to find a term t of type $(B \to B) \to B$ where $B = \{0, 1\}$ such that

$$t\ d_1 = 0, \qquad t\ d_2 = 1$$

where $d_1\ x = \neg\ x$ and $d_2\ x = x$. Furthermore this term should use only as constant 0 and 1.

Notice that we can enumerate all terms of type $(B \to B) \to B$: they are of the form $\lambda f\ f^k\ 0$ or $\lambda f\ f^k\ 1$. But to try them one after the other gives only a semi-decision procedure. It is quite remarkable that this question is actually decidable.

By the fundamental theorem of logical relations, *if* such a term exists we have $R(t\ d_1, t\ d_2) = R(0, 1)$ for any relation R such that

- $R(0,0)$ and $R(1,1)$,
- $R(x,y) \to R(d_1\ x, d_2\ y)$.

What is remarkable is that conversely, if we have

$$(R)[R(0,0)\rightarrow R(1,1)\rightarrow[(x)(y)R(x,y)\rightarrow R(d_1\ x,d_2\ y)]\rightarrow R(0,1)] \qquad (*)$$

then there exists a term t such that $t\ d_1 = 0$ and $t\ d_2 = 1$. This follows by taking for R the relation

$$\lambda x\lambda y\ \exists t\ [t\ d_1 = x \wedge t\ d_2 = y].$$

So (*) is actually a necessary and sufficient condition for the existence of a solution. It does not seem a priori algorithmic, since (*) is expressed with an universal quantification over all relations. Remark however that we can list all such relations. Furthermore, there is even a clever reading of (*) that gives an elegant algorithm to test (*): read the relation

$$\lambda x\lambda y\ (R)[R(0,0)\rightarrow R(1,1)\rightarrow[(x)(y)R(x,y)\rightarrow R(d_1\ x,d_2\ y)]\rightarrow R(x,y)]$$

as meaning that (x,y) is in the set of pairs generated by the rule

- $\rightarrow(0,0)$,
- $\rightarrow(1,1)$,
- $(x,y)\rightarrow(d_1\ x,d_2\ y)$.

Check that $(0,1)$ is in this set, and uses this derivation of $(0,1)$ to build t.

3.2 Adequacy Theorem

Here is still another application of the notion of logical relation, which is the theoretical foundation of the use of domain theory for proving equivalence of programs.

The adequacy theorem shows that there are good relations between the syntax and the semantics for ground type. It is stated here for an idealised programming language: simple typed λ-calculus + a fixed-point combinator Y. We assume that among the base types, there is a type N of natural numbers, and we take for its semantics the "flat" domain $D_N = \{\perp\}\cup N$.

The following Theorem holds for an *arbitrary* model of typed λ-calculus with fixed-point.

Theorem: let t be a closed term of type N, then $t()$ is convertible to an integer n iff its denotation $t()$ is equal to n.

One direction is the consistency of the semantics. The other direction is subtler: a priori, it may well be that the semantics "lies", it says for instance that $t = 1$ in the model though t diverges as a program. The proof is an application of logical relation. Define for each type σ a relation R_σ between the syntax and the semantics, i.e. between the given model V_σ and the domain model D_σ. We define $R_N(u,v)$ to mean that $v =\perp$ or else that there exists n such that $v = n$ and $u = n \in V_N$. At higher type $R_{\sigma\rightarrow\tau}(w,f)$ means that $R_\sigma(u,v)$ implies $R_\tau(w\ u,f\ v)$. We expect then that for all term t, ρ value environment for t and ρ' domain environment for t, if ρ and ρ' are related, then we have $R_\sigma(t\rho,t\rho')$, where σ is the type of t. This would imply the theorem when σ is N. By the fundamental theorem of logical relation, extended with constants, we are reduced to check that $R(Y,Y)$ holds.

Lemma: For each type σ σ, we have

- $R_\sigma(u,\perp)$ for any $u\in V_\sigma$,
- if v is the sup of the increasing chain (v_n), and $R(u,v_n)$ for each n, then $R(u,v)$.

The proof of this lemma is rather direct.

Corollary: We have $R(Y,Y)$ that is, if $R(u,v)$ then $R(Y\ u, \bigvee v^n\ \bot)$.

Proof: By the lemma, it is enough to show $R(Y\ u, v^n\ \bot)$ for all n. We show this by induction on n. The lemma shows the case $n=0$. If it is true for n, since $R(u,v)$ we deduce $R(u\ (Y\ u), v^{n+1}\ \bot)$. But $Y\ u$ is equal to $u\ (Y\ u)$, hence the result. $\Box$

This result is important in order to prove operational equivalence: let say that t_1 and t_2 programs of type σ (that may be functional) are *operationally equivalent* iff for all context $C[\_]$ of type N, $C[t_1]$ is convertible to an integer iff $C[t_2]$ is convertible to an integer, and in that case, these integers are equal. The intuition is that we can "test" programs only by embedding them into programs that are "observable", and functions, functionals... are not really observable (they are really ideal elements). A direct application of the theorem is the following result.

Corollary: Denotational equality implies operational equivalence.

The derivation of this corollary uses in an essential way the fact that the semantics is *compositional*: the semantics of an expression is a function of the semantics of its subpart. (This is reflected that the definition of the semantics of a term is by structural induction.) This corollary gives an elegant way of proving the operational equivalence of part of the programs.

Exercice. (Manna) Let $G : \sigma \rightarrow \sigma$ and $P : \sigma \rightarrow \mathsf{B}$ be arbitrary functions, and $F : \tau \rightarrow \tau$ a strict function. Define the function $H = Y\ K$, where

$$K = \lambda h, x, y \text{ if } P\ x \text{ then } y \text{ else } F\ (h\ (Gx)\ y).$$

Show $F\ (H\ x\ y) = H\ x\ (F\ y)$ for all x, y via denotational semantics. It follows that the programs $\lambda x, y\ F\ (H\ x\ y)$ and $\lambda x, y\ H\ x\ (F\ y)$ are operationally equivalent by the adequacy theorem.

4 Where does this come from?

It is extremely instructive to read Reynolds' papers [15, 16] to understand one motivation of the introduction of type variables and logical relations. One wants to represent the idea of type definitions and type abstractions. The idea is that one can define a type with some operations, for instance

$$\tau = \mathit{int}, \quad \mathit{mk} : \tau = 0, \quad \mathit{inc} : \tau \rightarrow \tau = \lambda x\ x+1, \quad \mathit{get} : \tau \rightarrow \mathit{int} = \lambda x\ x$$

and then uses the type τ and these operations $\mathit{mk}, \mathit{inc}, \mathit{get}$ but, and this is the important point, *without* having access to their definitions. This is represented quite naturally in polymorphic λ-calculus as a term of the form

$$(\lambda\ \tau(\lambda\ \mathit{mk} : \tau)(\lambda\ \mathit{inc} : \tau \rightarrow \tau)(\lambda\ \mathit{get} : \tau \rightarrow \mathit{int})\ \ e)\ \mathit{int}\ 0\ (\lambda x\ x+1)\ (\lambda x\ x)$$

This motivates the introduction of type variables. For motivating logical relations, suppose that we change the given implementation and we take instead

$$\tau = \mathit{int}, \quad \mathit{mk} : \tau = 0, \quad \mathit{inc} : \tau \rightarrow \tau = \lambda x\ x-1, \quad \mathit{get} : \tau \rightarrow \mathit{int} = \lambda x\ (-x)$$

We can prove using logical relation that if we have a term $e : \mathit{int}$ using τ, mk, inc, get as variables and we consider two close instantions e_1 and e_2 then we have $e_1 = e_2 : \mathit{int}$. In order to prove this, consider the language where we add a new type constant τ and news operations

$mk : \tau,\ inc : \tau{\rightarrow}\tau,\ get : \tau{\rightarrow}int$. A model of this language gives a meaning to the type τ and the operations mk, inc and get. We give two different models, with the logical relation given by $R_\tau(x,y))$ iff $x + y = 0$. Applying the fundamental theorem we get that for any related pairs of environment ρ_1, ρ_2 we have $R_\alpha(e\rho_1, e\rho_2)$. In particular if α does not mention τ we get $e\rho_1 = e\rho_2$.

For another example, consider the two implementations

$$\tau = bool,\quad bit : \tau = true,\quad flip : \tau{\rightarrow}\tau = not,\ \ read : \tau{\rightarrow}bool = \lambda x\ x$$

and

$$\tau = int,\quad bit : \tau = 1,\quad flip : \tau{\rightarrow}\tau = \lambda x\ \ -x,\ \ read : \tau{\rightarrow}bool = \lambda x\ x > 0$$

then these are two related different implementations of the same abstract structure

$$(\tau, a : \tau, f : \tau{\rightarrow}\tau, g : \tau{\rightarrow}bool).$$

Here we take $R(b,x)$ iff $b = true$ and $x > 0$ or $b = false$ and $x < 0$. Then $R(a_1, a_2)$ holds but also $R(f_1, f_2)$ and $R(g_1, g_2)$.

Notice that from the adequacy theorem we get as a corollary a purely operational result, that should be hard to prove in an operational way. This results also holds in presence of a fixed-point operator.

References

[1] P. Andrews. General Models, Descriptions, and Choice in Type Theory. Journal of Symbolic Logic, 37 (1972), p. 385-394.

[2] P. Andrews. General Models and Extensionality. Journal of Symbolic Logic, 37 (1972), p. 395-397.

[3] H. Barendregt. Typed Lambda Calculi. In Handbook of Logic in Computer Science, Abramsky et al eds, Oxford University Press, 1992

[4] A. Church. A Set of Postulates for the Foundation of Logic. Annals of Mathematics, 33 (1932), p. 346-366.

[5] A. Church. A formulation of the simple theory of types. Journal of Symbolic Logic, 5 (1940), p. 56-68.

[6] S. Fortune, D. Leivant and M. O'Donnell. The expressiveness of simple and second order lambda calculus. Journal of the ACM 30 (1983), p. 151-185.

[7] R.O. Gandy. On The Axiom of Extensionality, part I. Journal of Symbolic Logic, 21 (1956).

[8] J.R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda Calculus.* London Mathematical Society, London, 1986.

[9] S.C. Kleene. A Theory of Positive Integers in Formal Logic. Part I. American Journal of Mathematics, 57 (1935), p. 153-173.

[10] J.Y. Girard, Y. Lafont, P. Taylor. *Proof and Types.* Cambridge Tracts in Theoretical Computer Science, 7, 1989.

[11] L. Henkin. Completeness in the theory of types. Journal of Symbolic Logic, 15 (1950), p. 81-91.

[12] L. Henkin. The discovery of my completeness proof. Bulletin of Symbolic Logic, 2 (1996), p. 127-158

[13] S.C. Kleene and B. Rosser. The inconsistency of certain formal logics. Annals of Mathematics, 36 (1935), p. 630-636.

[14] J. C. Mitchell. Type systems for programming languages In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, p. 367-458, 1990.

[15] J.C. Reynolds. Towards a Theory of Type Structures. In: LNCS 19, p. 408-425.

[16] J.C. Reynolds. Types, abstraction and parametric polymorphism. In: Information Processing 83, edired by R.E.A. Mason. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1983, pp. 513 - 523.

[17] D. Scott. Domains and Logics, *extended abstract.* In Logic in Computer Science, 1989, p. 4-5.

[18] A. Stoughton. Mechanizing Logical Relations. LNCS 802, p. 359-377, 1994

[19] P. Wadler. Theorems for free! ACM Functional Programming Languages and Computer Architecture, 1989, 347-359.

[20] G. Winskel. *The Formal Semantics of Programming Languages, an Introduction.* MIT Press, 1993.