Logical Methods in Computer Science
Vol. 3 (4:12) 2007, pp. 1–16
www.lmcs-online.org

Submitted Mar. 4, 2007
Published Dec. 4, 2007

A PROOF OF STRONG NORMALISATION USING DOMAIN THEORY

THIERRY COQUAND [a] AND ARNAUD SPIWACK [b]

[a] Chalmers Tekniska Högskola, Gothenburg
e-mail address: coquand@cs.chamlers.se

[b] LIX, Ecole Polytechnique
e-mail address: Arnaud.Spiwack@lix.polytechnique.fr

Abstract. Ulrich Berger presented a powerful proof of strong normalisation using domains, in particular it simplifies significantly Tait's proof of strong normalisation of Spector's bar recursion. The main contribution of this paper is to show that, using ideas from intersection types and Martin-Löf's domain interpretation of type theory one can in turn simplify further U. Berger's argument. We build a domain model for an untyped programming language where U. Berger has an interpretation only for typed terms or alternatively has an interpretation for untyped terms but need an extra condition to deduce strong normalisation. As a main application, we show that Martin-Löf dependent type theory extended with a program for Spector double negation shift is strongly normalising.

Introduction

In 1961, Spector [23] presented an extension of Gödel's system T by a new schema of definition called bar recursion. With this new schema, he was able to give an interpretation of Analysis, extending Gödel's Dialectica interpretation of Arithmetic, and completing preliminary results of Kreisel [15]. Tait proved a normalisation theorem for Spector's bar recursion, by embedding it in a system with infinite terms [25]. In [9], an alternative form of bar recursion was introduced. This allowed to give an interpretation of Analysis by modified realisability, instead of Dialectica interpretation. The paper [9] presented also a normalisation proof for this new schema, but this proof, which used Tait's method of introducing infinite terms, was quite complex. It was simplified significantly by U. Berger [11, 12], who used instead a modification of Plotkin's computational adequacy theorem [19], and could prove *strong* normalisation. In a way, the idea is to replace infinite terms by elements of a domain interpretation. This domain has the property that a term is strongly normalisable if its semantics is $\neq\bot$

The main contribution of this paper is to show that, using ideas from intersection types [3, 6, 7, 18] and Martin-Löf's domain interpretation of type theory [16], one can in turn simplify further U. Berger's argument. Contrary to [11], we build a domain model for an

1998 ACM Subject Classification: F.4.1.

Key words and phrases: strong normalisation, λ-calculus, double-negation shift, Scott domain, λ-model, rewriting, denotational semantics.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-3 (4:12) 2007

untyped programming language. Compared to [12], there is no need of an extra hypothesis to deduce strong normalisation from the domain interpretation. A noteworthy feature of this domain model is that it is in a natural way a *complete* lattice, and in particular it has a *top* element which can be seen as the interpretation of a top-level exception in programming languages. We think that this model can be the basis of *modular* proofs of strong normalisation for various type systems. As a main application, we show that Martin-Löf dependent type theory extended with a program for Spector double negation shift [23][1], similar to bar recursion, has the strong normalisation property.

1. An Untyped Programming Language

Our programming language is untyped λ-calculus extended with constants, and has the following syntax.

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid c \mid f$$

There are two kinds of constants: *constructors* $c, c', \ldots$ and *defined constants* $f, g, \ldots$. We use $h, h', \ldots$ to denote a constant which may be a constructor or defined. Each constant has an *arity*, but can be partially applied. We write $\mathsf{FV}(M)$ for the set of free variables of M. We write $N(x = M)$ the result of substituting the free occurences of x by M in N and may write it $N[M]$ if x is clear from the context. We consider terms up to α-conversion.

The computation rules of our programming language are the usual β-reduction and ι-reduction defined by a set of rewrite rules of the form

$$f\ p_1 \ldots p_k = M$$

where k is the arity of f and $\mathsf{FV}(M) \subseteq \mathsf{FV}(f\ p_1 \ldots p_k)$. In this rewrite rule, $p_1, \ldots, p_k$ are *constructor patterns i.e.* terms of the form

$$p ::= x \mid c\ p_1 \ldots p_l$$

where l is the arity of c. Like in [11], we assume our system of constant reduction rules to be *left linear*, i.e. a variable occurs at most once in the left hand side of a rule, and *mutually disjoint*, i.e. the left hand sides of two disjoint rules are non-unifiable. We write $M \to M'$ if M reduces in one step to M' by β, ι-reduction and $M =_{\beta,\iota} M'$ if M, M' are convertible by β, ι conversion. It follows from our hypothesis on our system of reduction rules that β, ι-reduction is confluent [14]. We write $\to(M)$ for the set of terms M' such that $M \to M'$.

We work with a given set of constants, that are listed in section 3, but our arguments are general and make use only of the fact that the reduction system is left linear and mutually disjoint. We call UPL, for Untyped Programming Language, the system defined by this list of constants and ι-reduction rules. The goal of the next section is to define a domain model for UPL that has the property that M is strongly normalizing if $[\![M]\!] \neq \bot$.

[1]This is the schema $(\forall x.\neg\neg P(x)) \to \neg\neg\forall x.P(x)$. Spector [23] remarked that it is enough to add this schema to intuitionistic analysis in order to be able to interpret classical analysis via negative translation.

$$
\begin{array}{rcl}
\nabla \cap U & = & \nabla \\
c\ U_1 \dots U_k \cap c'\ V_1 \dots V_l & = & \nabla \\
c\ U_1 \dots U_k \cap V \to W & = & \nabla \\
(U \to V_1) \cap (U \to V_2) & = & U \to (V_1 \cap V_2) \\
c\ U_1 \dots U_k \cap c\ V_1 \dots V_k & = & c\ (U_1 \cap V_1) \dots (U_k \cap V_k)
\end{array}
$$

$$
\frac{U_1 \subseteq U_2 \quad U_2 \subseteq U_3}{U_1 \subseteq U_3}
\qquad
\frac{U_1 \subseteq V_1 \quad \dots \quad U_k \subseteq V_k}{c\ U_1 \dots U_k \subseteq c\ V_1 \dots V_k}
$$

$$
\frac{}{U \subseteq U}
\qquad
\frac{U \subseteq V_1 \quad U \subseteq V_2}{U \subseteq V_1 \cap V_2}
$$

$$
\frac{}{V_1 \cap V_2 \subseteq V_1}\ \frac{}{V_1 \cap V_2 \subseteq V_2}
\qquad
\frac{U_2 \subseteq U_1 \quad V_1 \subseteq V_2}{U_1 \to V_1 \subseteq U_2 \to V_2}
$$

Figure 1: Formal inclusion

2. A domain for strong normalization

2.1. **Formal Neighbourhoods.**

Definition 2.1. The *Formal Neighbourhoods* are given by the following grammar:

$$U, V ::= \nabla \mid c\ U_1 \dots U_k \mid U \to V \mid U \cap V$$

On these neighbourhoods we introduce a *formal inclusion* $\subseteq$ relation defined inductively by the rules of Figure 1. In these rules we use the formal equality relation $U = V$ defined to be $U \subseteq V$ and $V \subseteq U$. We let $\mathcal{M}$ be the set of neighbourhoods quotiented by the formal equality. The terminology "formal neighbourhoods" comes from [15, 21, 16].

Lemma 2.2. The formal inclusion and equality are both *decidable* relations, and $\mathcal{M}$ is a poset for the formal inclusion relation, and $\cap$ defines a binary meet operation on $\mathcal{M}$. We have $c\ U_1 \dots U_k \neq c'\ V_1 \dots V_l$ if $c \neq c'$ and $c\ U_1 \dots U_k = c\ V_1 \dots V_k$ if and only if $U_1 = V_1, \dots, U_k = V_k$. An element in $\mathcal{M}$ is either ∇ or of the form $c\ U_1 \dots U_k$ or of the form $(U_1 \to V_1) \cap \dots \cap (U_n \to V_n)$ and this defines a partition of $\mathcal{M}$. Furthermore the following "continuity condition" holds: if I is a (nonempty) finite set and $\bigcap_{i \in I}(U_i \to V_i) \subseteq U \to V$ then the set $J = \{i \in I \mid U \subseteq U_i\}$ is not empty and $\bigcap_{i \in J} V_i \subseteq V$. Note that there is no maximum element, where there usually is one. This is linked to the fact that we are aiming to prove *strong* normalisation, not weak normalisation.

Similar results are proved in [5, 3, 7, 6, 16].

Proof. We introduce the set of neighbourhoods in "normal form" by the grammar

$$
\begin{array}{lcl}
W, W' & ::= & \nabla \mid c\ W_1\ \dots\ W_k \mid I \\
I & ::= & (W_1 \to W_1') \cap \dots \cap (W_n \to W_n')
\end{array}
$$

and define directly the operation $\cap$ and the relation $\subseteq$ on this set. An element in normal form W is of the form ∇ or $c\ W_1\ \dots\ W_k$ or is a finite formal intersection $\cap X$ where X is a nonempty finite set of elements of the form $W \to W'$. The definition of $\cap$ and $\subseteq$ will be recursive, using the following complexity measure: $|\nabla| = 0$, $|c\ W_1\ \dots\ W_k| = 1 + max(|W_1|, \dots, |W_k|)$ and $|\cap_i (W_i \to W_i')| = 1 + max_i(|W_i|, |W_i'|)$.

We define

$\nabla \cap W = W \cap \nabla = \nabla$
$c\ W_1\ \ldots\ W_k \cap c\ W'_1\ \ldots\ W'_k = c\ (W_1 \cap W'_1)\ \ldots\ (W_k \cap W'_k)$
$c\ W_1\ \ldots\ W_k \cap c'\ W'_1\ \ldots\ W'_l = \nabla$
$c\ W_1\ \ldots\ W_k \cap (\cap X) = (\cap X) \cap c\ W_1\ \ldots\ W_k = \nabla$
$(\cap X) \cap (\cap Y) = \cap(X \cup Y)$.

Notice that we have $|W_1 \cap W_2| \leq max(|W_1|, |W_2|)$.

We have furthermore $\nabla \subseteq W$ and $c\ W_1\ \ldots\ W_k \cap c\ W'_1\ \ldots\ W'_k$ iff $W_i \subseteq W'_i$ for all i and finally $\cap X \subseteq \cap Y$ iff for all $W \to W'$ in Y there exists $W_1 \to W'_1, \ldots, W_k \to W'_k$ in X such that $W \subseteq W_1, \ldots, W \subseteq W_k$ and $W'_1 \cap \cdots \cap W'_k \subseteq W'$. This definition is well founded since $|W'_1 \cap \cdots \cap W'_k| < |\cap X|$ and $|W'| < |\cap Y|$. One can then prove that relation $\subseteq$ and the operation $\cap$ satisfies all the laws of Figure 1 on the set of neighbourhoods of complexity $< n$ by induction on n.

Since all the laws of Figure 1 are valid for this structure we get in this way a concrete representation of the poset $\mathcal{M}$, and all the properties of this poset can be directly checked on this representation. □

We associate to $\mathcal{M}$ a type system defined in Figure 2 (when unspecified, k is the arity of the related constant). It is a direct extension of the type systems considered in [3, 5, 6, 7, 16]. The typing rules for the constructors and defined constants appear to be new however. Notice that the typing of the function symbols is very close to a recursive definition of the function itself. Also, we make use of the fact that, as a consequence of Lemma 2.2, one can define when a constructor pattern matches an element of $\mathcal{M}$.

Lemma 2.3. If $\Gamma \vdash_{\mathcal{M}} \lambda x.N : U$ then there exists a family U_i, V_i such that $\Gamma, x : U_i \vdash_{\mathcal{M}} N : V_i$ and $\cap_i(U_i \to V_i) \subseteq U$.

Proof. Direct by induction on the derivation. □

Lemma 2.4. If $\Gamma \vdash_{\mathcal{M}} \lambda x.N : U \to V$ then $\Gamma, x{:}U \vdash_{\mathcal{M}} N : V$.

Proof. We have a family U_i, V_i such that $\Gamma, x : U_i \vdash_{\mathcal{M}} N : V_i$ and $\cap_i(U_i \to V_i) \subseteq U \to V$. By Lemma 2.2 there exists $i_1, \ldots, i_k$ such that $U \subseteq U_{i_1}, \ldots, U \subseteq U_{i_k}$ and $V_{i_1} \cap \cdots \cap V_{i_k} \subseteq V$. This together with $\Gamma, x : U_i \vdash_{\mathcal{M}} N : V_i$ imply $\Gamma, x : U \vdash_{\mathcal{M}} N : V$. □

Lemma 2.5. If $\Gamma \vdash_{\mathcal{M}} N\ M : V$ then there exists U such that $\Gamma \vdash_{\mathcal{M}} N : U \to V$ and $\Gamma \vdash_{\mathcal{M}} M : U$.

Proof. Direct by induction on the derivation. □

2.2. **Reducibility candidates.**

Definition 2.6. $\mathcal{S}$ (the set of simple terms) is the set of terms that are neither an abstraction nor a constructor headed term, nor a partially applied destructor headed term (*i.e.* $f\ M_1\ \ldots\ M_n$ is simple if n is greater or equal to the arity of f).

Definition 2.7. A *reducibility candidate* X is a set of terms with the following properties:

(CR1): $X \subseteq \mathsf{SN}$
(CR2): $\to(M) \subseteq X$ if $M \in X$
(CR3): $M \in X$ if $M \in \mathcal{S}$ and $\to(M) \subseteq X$

$$\frac{x : U \in \Gamma}{\Gamma \vdash_{\mathcal{M}} x : U}$$

$$\frac{}{\Gamma \vdash_{\mathcal{M}} c : U_1 \to \ldots \to U_k \to c\ U_1 \ldots U_k}$$

$$\frac{\Gamma, x{:}U \vdash_{\mathcal{M}} M : V}{\Gamma \vdash_{\mathcal{M}} \lambda x.M : U \to V}$$

$$\frac{\Gamma \vdash_{\mathcal{M}} N : U \to V \quad \Gamma \vdash_{\mathcal{M}} M : U}{\Gamma \vdash_{\mathcal{M}} N\ M : V}$$

$$\frac{\Gamma \vdash_{\mathcal{M}} M : U \quad \Gamma \vdash_{\mathcal{M}} M : V}{\Gamma \vdash_{\mathcal{M}} M : U \cap V}$$

$$\frac{\Gamma \vdash_{\mathcal{M}} M : V \quad V \subseteq U}{\Gamma \vdash_{\mathcal{M}} M : U}$$

$$\frac{\begin{array}{c} f\ p_1 \ldots p_k = M \quad p_i(W_1, \ldots, W_n) = U_i \\ \Gamma, x_1{:}W_1, \ldots, x_n{:}W_n \vdash_{\mathcal{M}} M : V \end{array}}{\Gamma \vdash_{\mathcal{M}} f : U_1 \to \ldots \to U_k \to V}$$

$$\frac{\begin{array}{c} \text{for any } U_1, \ldots, U_k \text{ such that} \\ \text{no rewrite rule of } f \text{ matches } U_1, \ldots, U_k \end{array}}{\Gamma \vdash_{\mathcal{M}} f : U_1 \to \ldots \to U_k \to \nabla}$$

Figure 2: Types with intersection in $\mathcal{M}$

It is clear that the reducibility candidates form a complete lattice w.r.t. the inclusion relation. In particular, there is a *least* reducibility candidate R_0, which can be inductively defined as the set of terms $M \in \mathcal{S}$ such that $\to(M) \subseteq R_0$. For instance, if M is a variable x, then we have $M \in R_0$ since $M \in \mathcal{S}$ and $\to(M) = \emptyset$.

We define two operations on sets of terms, which preserve the status of candidates. If c is a constructor of arity k and $X_1, \ldots, X_k$ are sets of terms then the set $c\ X_1 \ldots X_k$ is inductively defined to be the set of terms M of the form $c\ M_1 \ldots M_k$, with $M_1 \in X_1 \ldots M_k \in X_k$ or such that $M \in \mathcal{S}$ and $\to(M) \subseteq c\ X_1 \ldots X_k$. If X and Y are sets of terms, $X \to Y$ is the set of terms N such that $N\ M \in Y$ if $M \in X$.

Lemma 2.8. If X and Y are reducibility candidates then so are $X \cap Y$ and $X \to Y$. If $X_1, \ldots, X_k$ are reducibility candidates then so is $c\ X_1 \ldots X_k$.

Definition 2.9. The function $[-]$ associates a reducibility candidate to each formal neighbourhood.

- $[\nabla] \triangleq R_0$
- $[c\ U_1 \ldots U_k] \triangleq c\ [U_1] \ldots [U_k]$
- $[U \to V] \triangleq [U] \to [V]$
- $[U \cap V] \triangleq [U] \cap [V]$

Lemma 2.10. If $U \subseteq V$ for the formal inclusion relation then $[U] \subseteq [V]$ as sets of terms.

This follows from the fact that all the rules of Figure 1 are valid for reducility candidates.

Theorem 2.11. If $\vdash_{\mathcal{M}} M : U$ then $M \in [U]$. In particular M is strongly normalising.

As usual, we prove that if $x_1 : U_1, \ldots, x_n : U_n \vdash_{\mathcal{M}} M : U$ and $M_1 \in [U_1], \ldots, M_n \in [U_n]$ then $M(x_1 = M_1, \ldots, x_n = M_n) \in [U]$. This is a mild extention of the usual induction on derivations. We sketch the extra cases:

- Subtyping: direct from Lemma 2.10.
- Constructor: direct from the definition of $[c\ U_1 \ldots U_k]$.
- Defined constant (case with a rewrite rule): we need a small remark: since $c'\ M_1 \ldots M_l \notin \mathcal{S}$ for any l, we have that $c'\ M_1 \ldots M_l \in c\ X_1 \ldots X_k$ implies $c' = c$ and $l = k$ by definition of $c\ X_1 \ldots X_k$. Knowing this we get that if $N_i \in p_i([W_1], \ldots, [W_n]))$, then $f\ N_1 \ldots N_k$ can only interract with one rewrite rule (remember that there is no critical pair). The definition of $c\ X_1 \ldots X_k$ also tells us that if the N_i are equal to $p_i(M_1, \ldots, M_n)$, then $M_j \in W_j$. From this the result follows easily.
- Defined constant (case with no rewrite rule): we need the same remark as in the previous case: $c'\ M_1 \ldots M_l \in c\ X_1 \ldots X_k$ implies that $c' = c$ and $l = k$. Additionally, $[\nabla]$ does not contain any constructor-headed term (since $[\nabla] \subseteq \mathcal{S}$). A consequence of these two remarks is that there cannot be any fully applied constructor-headed term in $[U \to V]$, by simple induction. In particular there is no term matched by a pattern in $[U \to V]$. Thus, since there is no rule matching the $U_1, \ldots, U_k$, we know that for any $N_1 \in [U_1], \ldots, N_k \in [U_k]$, $f\ N_1 \ldots N_k$ is not matched by any rewrite rule; it is, however, a simple term. It follows easily that $f\ N_1 \ldots N_k \in [\nabla]$. □

2.3. **Filter Domain.**

Definition 2.12. An *I-filter*[2] over $\mathcal{M}$ is a subset $\alpha \subseteq \mathcal{M}$ with the following closure properties:

- if $U, V \in \alpha$ then $U \cap V \in \alpha$
- if $U \in \alpha$ and $U \subseteq V$ then $V \in \alpha$

It is clear that the set D of all I-filters over $\mathcal{M}$ ordered by the set inclusion is a complete algebraic domain. The finite elements of D are exactly $\emptyset$ and the principal I-filters $\uparrow U \triangleq \{V \mid U \subseteq V\}$. The element $\top = \uparrow \nabla$ is the greatest element of D and the least element is $\bot = \emptyset$.

We can define on D a binary application operation

$$\alpha\ \beta \triangleq \{V \mid \exists U, U \to V \in \alpha \wedge U \in \beta\}$$

We have always $\alpha\ \bot = \bot$ and $\top\ \beta = \top$ if $\beta \neq \bot$. We write $\alpha_1 \ldots \alpha_n$ for $(\ldots(\alpha_1\ \alpha_2)\ \ldots)\ \alpha_n$.

[2]This terminology, coming from [6], stresses the fact that the empty set is also an I-filter.

2.4. **Denotational semantics of UPL.** As usual, we let $\rho, \nu, \ldots$ range over *environments*, i.e. mapping from variables to D.

Definition 2.13. If M is a term of UPL, $[\![M]\!]_\rho$ is the I-filter of neighbourhoods U such that $x_1{:}V_1, \ldots, x_n{:}V_n \vdash_{\mathcal{M}} M : U$ for some $V_i \in \rho(x_i)$ with $\mathsf{FV}(M) = \{x_1, \ldots, x_n\}$.

A direct consequence of this definition and of Theorem 2.11 is then

Theorem 2.14. If there exists ρ such that $[\![M]\!]_\rho \neq \bot$ then M is strongly normalising.

Notice also that we have $[\![M]\!]_\rho = [\![M]\!]_\nu$ as soon as $\rho(x) = \nu(x)$ for all $x \in \mathsf{FV}(M)$. Because of this we can write $[\![M]\!]$ for $[\![M]\!]_\rho$ if M is closed. If c is a constructor, we write simply c for $[\![c]\!]$.

Lemma 2.15. We have $c\ \alpha_1 \ldots \alpha_k \neq c'\ \beta_1 \ldots \beta_l$ if $c \neq c'$ and $c\ \alpha_1 \ldots \alpha_k = c\ \beta_1 \ldots \beta_k$ if and only if $\alpha_1 = \beta_1 \ldots \alpha_k = \beta_k$, whenever $\alpha_i \neq \bot$, $\beta_j \neq \bot$. An element of D is either $\bot$, or $\top$ or of the form $c\ \alpha_1\ \ldots\ \alpha_k$ with c of arity k and $\alpha_i \neq \bot$ or is a sup of elements of the form $\uparrow (U \to V)$. This defines a partition of D.

Proof. Follows from Lemma 2.2. □

As a consequence of Lemma 2.15, it is possible to define when a constructor pattern matches an element of D. The next result expresses the fact that we have defined in this way a *strict model* of UPL.

Theorem 2.16.

$$\begin{array}{lcll} [\![x]\!]_\rho & = & \rho(x) & \\ [\![N\ M]\!]_\rho & = & [\![N]\!]_\rho\ [\![M]\!]_\rho & \\ [\![\lambda x.M]\!]_\rho\ \alpha & = & [\![M]\!]_{(\rho, x:=\alpha)} & \text{if } \alpha \neq \bot \end{array}$$

If $f\ p_1\ \ldots\ p_k = M$ and $\alpha_i = [\![p_i]\!]_\rho$ then $[\![f]\!]\ \alpha_1 \ldots \alpha_k = [\![M]\!]_\rho$. If there is no rule for f which matches $\alpha_1, \ldots, \alpha_k$ and $\alpha_1, \ldots, \alpha_k$ are $\neq \bot$ then $[\![f]\!]\ \alpha_1 \ldots \alpha_k = \top$. Finally, if for all $\alpha \neq \bot$ we have $[\![M]\!]_{(\rho, x:=\alpha)} = [\![N]\!]_{(\nu, y:=\alpha)}$ then $[\![\lambda x.M]\!]_\rho = [\![\lambda y.N]\!]_\nu$.

Proof. The second equality follows from Lemma 2.5 and the third equality follows from Lemma 2.4. □

Corollary 2.17. $[\![N(x = M)]\!]_\rho = [\![N]\!]_{(\rho, x=[\![M]\!]_\rho)}$

3. Application to Spector's Double Negation Shift

The goal of this section is to prove strong normalisation for dependent type theory extended with Spector's double negation shift [23]. The version of type theory we present is close to the one in [17]: we have a type of natural numbers $\mathsf{Nat} : \mathsf{U}$, where U is an universe. It is shown in [17], using the propositions-as-types principle, how to represent intuitionistic higher-order arithmetic in type theory. It is then possible to formulate Spector's double negation shift as

$$(\Pi n : \mathsf{Nat}.\neg\neg B\ n) \to \neg\neg \Pi n : \mathsf{Nat}.B\ n$$

where $\neg A$ is an abreviation for $A \to \mathsf{N}_0$ and $B : \mathsf{Nat} \to \mathsf{U}$. Spector showed [23] that it is enough to add this schema (Axiom F in [23]) to intuitionistic analysis in order to be able to interpret classical analysis via a negative translation. We show how to extend dependent type theory with a constant of this type in such a way that strong normalisation is preserved.

It follows then from [23] that the proof theoretic strength of type theory is much stronger with this constant and has the strength of classical analysis.

3.1. **General Rules of Type Theory.** We have a constructor Fun of arity 2 and we write $\Pi x{:}A.B$ instead of $\mathsf{Fun}\ A\ (\lambda x.B)$, and $A \to B$ instead of $\mathsf{Fun}\ A\ (\lambda x.B)$ if x is not free in B. We have a special constant U for universe. (We recall that we consider terms up to α-conversion.) A *context* is a sequence $x_1 : A_1, \ldots, x_n : A_n$, where the x_i are pairwise distinct.

They are three forms of judgements

$$\Gamma \vdash A \quad \Gamma \vdash M : A \quad \Gamma \vdash$$

The last judgement $\Gamma \vdash$ expresses that Γ is a well-typed context. We may write $J\ [x : A]$ for $x : A \vdash J$.

The typing rules are in figure 3.1

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A} \qquad \frac{\Gamma, x : A \vdash B}{\Gamma \vdash \Pi x{:}A.B}$$

$$\frac{(x : A) \in \Gamma \quad \Gamma \vdash}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash N : \Pi x{:}A.B \quad \Gamma \vdash M : A}{\Gamma \vdash N\ M : B[M]}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B \quad A =_{\beta,\iota} B}{\Gamma \vdash M : B}$$

We express finally that the universe U is closed under the product operation.

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma, x : A \vdash B : \mathsf{U}}{\Gamma \vdash \Pi x{:}A.B : \mathsf{U}}$$

Figure 3: Typing Rules of Type Theory

The constants are the ones of our language UPL, described in the next subsection.

3.2. **Specific Rules.** We describe here both the untyped language UPL (which will define the ι reduction) and the fragment of type theory that we need in order to express a program for Spector double negation shift. The constant of form (op) are used as infix operators.

The constructors are $\mathsf{U}, \mathsf{Nat}, \mathsf{N}_0, \mathsf{N}_1, \mathsf{0}$ (arity 0), $\mathsf{S}, \mathsf{Inl}, \mathsf{Inr}$ (arity 1) and $(+), (\times), \mathsf{Fun}, \mathsf{Pair}$ (arity 2). To define the domain D as in the previous sections, it is enough to know these constructors.

The defined constants of the language UPL are $\mathsf{vec}, \mathsf{get}, \mathsf{trim}, T, \mathsf{head}, \mathsf{tail}, (\leq), \mathsf{less}, \mathsf{Rec}, \neg, \mathsf{exit}, \Phi, \Psi$. The arities are clear from the given ι-rules. From these ι-rules it is then possible to interpret each of these constants as an element of the domain D.

At the same time we introduce these constants (constructors or defined constants) we give their intended types.

First we have the type of natural numbers Nat with two constructors:

$$\begin{array}{l}\mathsf{Nat} : \mathsf{U} \\ \| \ \mathsf{0} : \mathsf{Nat} \\ \| \ \mathsf{S} : \mathsf{Nat}\end{array}$$

We also add the natural number recursor Rec so that the language contains Heyting airthmetic:

$$\begin{array}{l}\mathsf{Rec} : C\ \mathsf{0} \to (\Pi n : \mathsf{Nat}.C\ n \to C\ (\mathsf{S}\ n)) \to \Pi n : \mathsf{Nat}.C\ n [C : \mathsf{Nat} \to \mathsf{U}] \\ \| \ \mathsf{Rec}\ P\ Q\ \ \mathsf{0}\ \ = \ N \\ \| \ \mathsf{Rec}\ P\ Q\ (\mathsf{S}\ x) = \ M\ x\ (\mathsf{Rec}\ N\ M\ x)\end{array}$$

In addition we add type connectives. $(+)$ stands for the type disjunction, and $(\times)$ for the pair type:

$$\begin{array}{l}(+) : \mathsf{U} \to \mathsf{U} \to \mathsf{U} \\ \| \ \mathsf{Inl}\ : \quad A \to A + B\ [A, B : \mathsf{U}] \\ \| \ \mathsf{Inr}\ : \quad B \to A + B\ [A, B : \mathsf{U}]\end{array}$$

$$\begin{array}{l}(\times) : \mathsf{U} \to \mathsf{U} \to \mathsf{U} \\ \| \ \mathsf{Pair} : A \to B \to A \times B\ [A, B : \mathsf{U}]\end{array}$$

We write (x, y) instead of $\mathsf{Pair}\ x\ y$, and $(x_1, \ldots, x_n)$ for $(\ldots (x_1, x_2), \ldots, x_n)$.

We also need the empty type N_0 (with no constructor):

$$\mathsf{N}_0 : \mathsf{U}$$

with which we can define exit, its elimination rule, also known as *ex falsum quod libet* and the negation $\neg$:

$$\mathsf{exit} : \mathsf{N}_0 \to A\ [A : \mathsf{U}]$$

$$\begin{array}{l}\neg : \mathsf{U} \to \mathsf{U} \\ \| \ \neg\ A \ = \ A \to \mathsf{N}_0\end{array}$$

Notice that the constant exit has no computation rule.

The last type we need to define is N_1, the unit type (*i.e.* with only one trivial constructor), in other word the type "true":

$$\begin{array}{l}\mathsf{N}_1 : \mathsf{U} \\ \| \ \mathsf{0} : \mathsf{N}_1\end{array}$$

Notice that $\mathsf{0}$ is polymorphic and is a constructor of both N_1 and Nat.

We can now start defining the more specific functions of our language. First comes $(\leq)$. It decides if its first argument is less or equal to its second one. Note that it returns either N_1 or N_0 which are types. This is an example of strong elimination, *i.e* defining a predicate using a recursive function.

$$\begin{array}{l}(\leq) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{U} \\ \| \ \ \mathsf{0} \ \leq \ n \ \ = \ \mathsf{N}_1 \\ \| \ (\mathsf{S}\ x) \leq \ \mathsf{0} \ \ = \ \mathsf{N}_0 \\ \| \ (\mathsf{S}\ x) \leq (\mathsf{S}\ n) \ = \ x \leq n\end{array}$$

Consequently we have the function less which proves essentially that $(\leq)$ is a total ordering:

$$\begin{array}{l}\mathsf{less} : \Pi x : \mathsf{Nat}.\Pi n : \mathsf{Nat}.(\mathsf{S}\ x \leq n) + (n \leq x)\\ \left\| \begin{array}{lllll} \mathsf{less} & x & 0 & = & \mathsf{Inr}\ 0\\ \mathsf{less} & 0 & (\mathsf{S}\ n) & = & \mathsf{Inl}\ 0\\ \mathsf{less} & (\mathsf{S}\ x) & (\mathsf{S}\ n) & = & \mathsf{less}\ x\ n \end{array}\right.\end{array}$$

In order to write the proof of the shifting rule it is convenient to have a type of vectors $\mathsf{vec}\ B\ n$, which is intuitively $(\dots(\mathsf{N}_1 \times B\ 0)\dots) \times B\ (n-1)$ and an access function of type $\Pi n : \mathsf{Nat}.\Pi x : \mathsf{Nat}.(S\ x \leq n) \to \mathsf{vec}\ B\ n \to B\ x$

Notice that this access function requires as an extra argument a proof that the index access is in the right range. To have such an access function is a nice exercise in programming with dependent types.

This has to be seen as the type of finite approximations of proofs of $\Pi n : \mathsf{Nat}.B\ n$. And the access function is the respective elimination rule (*i.e.* a finite version of the forall elimination rule of natural deduction).

The type of vectors vec is defined recursively

$$\begin{array}{l}\mathsf{vec} : (\mathsf{Nat} \to \mathsf{U}) \to \mathsf{Nat} \to \mathsf{U}\\ \left\| \begin{array}{llll} \mathsf{vec}\ B & 0 & = & \mathsf{N}_1\\ \mathsf{vec}\ B\ (\mathsf{S}\ x) & & = & (\mathsf{vec}\ B\ x) \times B\ x \end{array}\right.\end{array}$$

With vec come two simple functions head and tail accessing respectively the two component of the pair (any non-0-indexed vector is a pair of an "element" and a shorter vector):

$$\begin{array}{l}\mathsf{head} : \Pi x : \mathsf{Nat}.(\mathsf{vec}\ B\ (\mathsf{S}\ x)) \to B\ x\\ \left\| \mathsf{head}\ x\ (v,u) = \ u\right.\end{array}$$

$$\begin{array}{l}\mathsf{tail} : \Pi x : \mathsf{Nat}.(\mathsf{vec}\ B\ (\mathsf{S}\ x)) \to \mathsf{vec}\ B\ x\\ \left\| \mathsf{tail}\ \ x\ (v,u) = \ v\right.\end{array}$$

In order to build the access function for type vec (which is supposed to extract the element of type $B\ x$ from a vector of a length longer than x) we introduce a function trim which shortens a vector of type $\mathsf{vec}\ B\ n$ into a vector of type $\mathsf{vec}\ B\ x$ by removing the $n-x$ first elements. The reason why such a function is useful is because we are trying to read the vector from the inside to the outside.

$$\begin{array}{l}T : (\mathsf{Nat} \to \mathsf{U}) \to \mathsf{U}\\ \left\| T\ P \ = \ \Pi k : \mathsf{Nat}.P\ (\mathsf{S}\ k) \to P\ k\right.\end{array}$$

$$\begin{array}{l}\mathsf{trim} : \Pi n : \mathsf{Nat}.\Pi m : \mathsf{Nat}.(n \leq m) \to \Pi P : \mathsf{Nat} \to \mathsf{U}.T\ P \to P\ m \to P\ n\\ \left\| \begin{array}{llllll} \mathsf{trim} & 0 & 0 & p\ P\ h\ v & = & v\\ \mathsf{trim} & 0 & (\mathsf{S}\ m) & p\ P\ h\ v & = & \mathsf{trim}\ 0\ m\ P\ h\ (h\ m\ v)\\ \mathsf{trim} & (\mathsf{S}\ n) & 0 & p\ P\ h\ v & = & \mathsf{exit}\ p\\ \mathsf{trim} & (\mathsf{S}\ n) & (\mathsf{S}\ m) & p\ P\ h\ v & = & \mathsf{trim}\ n\ m\ p\ (\lambda x.P\ (\mathsf{S}\ x))\ (\lambda x.h\ (\mathsf{S}\ x))\ v \end{array}\right.\end{array}$$

As a consequence of the function trim we can define in a rather simple way the access function get:

$$\begin{array}{l}\mathsf{get} : \Pi B : \mathsf{Nat} \to \mathsf{U}.\Pi n : \mathsf{Nat}.\Pi x : \mathsf{Nat}.(\mathsf{S}\ x \leq n) \to \mathsf{vec}\ B\ n \to B\ x\\ \left\| \mathsf{get}\ B\ n\ x\ p\ v \ = \ \mathsf{head}\ x\ (\mathsf{trim}\ (\mathsf{S}\ x)\ n\ p\ (\mathsf{vec}\ B)\ \mathsf{tail}\ v)\right.\end{array}$$

We need the following result on the domain interpretation of this function get. To simplify the notations we write h instead of $[\![h]\!]$ if h is a constant of the language. We also write $\overline{l}$ for $\mathsf{S}^l\ 0$.

Lemma 3.1. Let $v \neq \bot, y \neq \bot$ and B such that for any l, $B\ (S^l\ \top) \neq \bot$ and $B\ \overline{l} \neq \bot$ (in particular, $B \neq \bot$). If $x = \overline{q}$ with $q < p$ then $\mathsf{get}\ \overline{p}\ x\ 0\ v = \mathsf{get}\ \overline{p+1}\ x\ 0\ (v, y)$. If $x = \mathsf{S}^q\ \top$ with $q < p$ then $\mathsf{get}\ \overline{p}\ x\ 0\ v = \top$.

Proof. Let us prove that if $x = \overline{q}$ with $q < p$ then $\mathsf{get}\ \overline{p}\ x\ 0\ v = \mathsf{get}\ \overline{p+1}\ x\ 0\ (v, y)$. The proof of the second part of the Lemma is similar. It is proved by the following sequence of propositions

- If $h = [\![\lambda x.f\ (\mathsf{S}\ x)]\!]_{(f=h)} \neq \bot$ and $h\ m\ u = h\ \top\ u$ for any m, u, $q \leq p$, $t \neq \bot$, $v \neq \bot$ and $P\ (\mathsf{S}^l\ \top) \neq \bot$ for any l (in particular, $P \neq \bot$), then $\mathsf{trim}\ \overline{q}\ \overline{p}\ t\ P\ v = (h\ \top)^{p-q}\ v$.
  This is proved by simple induction on q and p. Using the definition of trim together with Theorem 2.16 and the fact that $P\ (\mathsf{S}^l\ \top) \neq \bot$ implies that $[\![\lambda f.f\ (\mathsf{S}\ x)]\!]_{(f=P)}\ (\mathsf{S}^l\ \top) = P\ (\mathsf{S}^{l+1}\ \top) \neq \bot$ for any l.
- $\mathsf{tail} = [\![\lambda x.f\ (\mathsf{S}\ x)]\!]_{(f=\mathsf{tail})} \neq \bot$ and $\mathsf{tail}\ m\ u = \mathsf{tail}\ \top\ u$. By Theorem 2.16.
- If $B\ (S^l\ \top) \neq \bot$ and $B\ \overline{l} \neq \bot$, then for all l $\mathsf{vec}\ B\ (\mathsf{S}^l\ \top) \neq \bot$. It is direct by induction on l using the definition of vec and Theorem 2.16.
- Finally

$$\begin{aligned}
\mathsf{get}\ \overline{p+1}\ x\ 0\ (v, y) &= \mathsf{head}\ x\ (\mathsf{trim}\ (\mathsf{S}\ x)\ \overline{p+1}\ 0\ (\mathsf{vec}\ B)\ \mathsf{tail}\ (v, y)) \\
&= \mathsf{head}\ x\ ((\mathsf{tail}\ \top)^{p-q}\ (v, y)) \\
&= \mathsf{head}\ x\ ((\mathsf{tail}\ \top)^{p-q-1}\ v) \\
&= \mathsf{head}\ x\ (\mathsf{trim}\ (\mathsf{S}\ x)\ \overline{p}\ 0\ (\mathsf{vec}\ B)\ \mathsf{tail}\ v) \\
&= \mathsf{get}\ \overline{p}\ x\ 0\ v
\end{aligned}$$

□

We can now introduce two functions Φ and Ψ, defined in a mutual recursive way. They define a slight generalisation of the double negation shift:

$$\Phi : \Pi B : \mathsf{Nat} \to \mathsf{U}.(\Pi n : \mathsf{Nat}.\neg\neg B\ n) \to \neg(\Pi n : \mathsf{Nat}.B\ n) \to \Pi n : \mathsf{Nat}.\neg\mathsf{vec}\ B\ n$$
$$\Psi : \Pi B : \mathsf{Nat} \to \mathsf{U}.(\Pi n : \mathsf{Nat}.\neg\neg B\ n) \to \neg(\Pi n : \mathsf{Nat}.B\ n) \to$$
$$\Pi n : \mathsf{Nat}.\mathsf{vec}\ B\ n \to \Pi x : \mathsf{Nat}.(\mathsf{S}\ x \leq n) + (n \leq x) \to B\ x$$

$$\begin{aligned}
&\Phi\ B\ H\ K\ n\ v &&= K\ (\lambda x.\Psi\ B\ H\ K\ n\ v\ x\ (\mathsf{less}\ x\ n)) \\
&\Psi\ B\ H\ K\ n\ v\ x\ (\mathsf{Inl}\ p) &&= \mathsf{get}\ B\ n\ x\ p\ v \\
&\Psi\ B\ H\ K\ n\ v\ x\ (\mathsf{Inr}\ p) &&= \mathsf{exit}\ (H\ n\ (\lambda y.\Phi\ B\ H\ K\ (\mathsf{S}\ n)\ (v, y)))
\end{aligned}$$

The program that proves Spector's double negation shift

$$\Pi B : \mathsf{Nat} \to \mathsf{U}.(\Pi n : \mathsf{Nat}.\neg\neg B\ n) \to \neg\neg(\Pi n : \mathsf{Nat}.B\ n)$$

is then $\lambda B.\lambda H.\lambda K.\Phi\ B\ H\ K\ 0\ 0$.

4. Model of type theory and strong normalisation

4.1. **Model.** We let $\mathsf{Pow}(\mathsf{D})$ be the collection of all subsets of D. If $X \in \mathsf{Pow}(\mathsf{D})$ and $F : X \to \mathsf{Pow}(\mathsf{D})$ we define $\Pi(X, F) \in \mathsf{Pow}(\mathsf{D})$ by $v \in \Pi(X, F)$ if and only if $u \in X$ implies $v\ u \in F(u)$.

A *totality predicate* on D is a subset X such that $\bot \notin X$ and $\top \in X$. We let $\mathsf{TP}(\mathsf{D})$ be the collection of all totality predicates.

Lemma 4.1. If $X \in \mathsf{TP}(\mathsf{D})$ and $F : X \to \mathsf{TP}(\mathsf{D})$ then $\Pi(X, F) \in \mathsf{TP}(\mathsf{D})$.

Proof. We have $\top \in X$. If $v \in \Pi(X, F)$ then $v\ \top \in F(\top)$ and so $v\ \top \neq \bot$ and $v \neq \bot$ hold. If $u \in X$ then $u \neq \bot$ so that $\top\ u = \top \in F(u)$. This shows $\top \in \Pi(X, F)$. □

Definition 4.2. A *model* of type theory is a pair T, El with $T \in \mathsf{TP}(\mathsf{D})$ and $El : T \to \mathsf{TP}(\mathsf{D})$ satisfying the property: if $A \in T$ and $u \in El(A)$ implies $F\ u \in T$ then $\mathsf{Fun}\ A\ F \in T$. Furthermore $El(\mathsf{Fun}\ A\ F) = \Pi(El(A), \lambda u.El(F\ u))$.

If we have a collection of constants with typing rules $\vdash h : A$ we require also $[\![A]\!] \in T$ and $[\![h]\!] \in El([\![A]\!])$.

Finally, for a model of type theory with universe U we require also: $\mathsf{U} \in T$, $El(\mathsf{U}) \subseteq T$ and $\mathsf{Fun}\ A\ F \in El(\mathsf{U})$ if $A \in El(\mathsf{U})$ and $F\ u \in El(\mathsf{U})$ for $u \in El(A)$.

The intuition is the following: $T \subseteq \mathsf{D}$ is the collection of elements representing types and if $A \in T$ the set $El\ A$ is the set of elements of type A. The first condition expresses that T is closed under the dependent product operation. The last condition expresses that U is a type and that $El\ (\mathsf{U})$ is a subset of T which is also closed under the dependent product operation.

The next result states the soundness of the semantics w.r.t. the type system.

Theorem 4.3. Let Δ be a context. Assume that $[\![A]\!]_\rho \in T$ and $\rho(x) \in El([\![A]\!]_\rho)$ for $x{:}A$ in Δ. If $\Delta \vdash A$ then $[\![A]\!]_\rho \in T$. If $\Delta \vdash M{:}A$ then $[\![A]\!]_\rho \in T$ and $[\![M]\!]_\rho \in El([\![A]\!]_\rho)$.

Proof. Direct by induction on derivations, using Theorem 2.16 and Corollary 2.17. For instance, we justify the application rule. We have by induction $[\![N]\!]_\rho \in \mathsf{El}\ (\mathsf{Fun}\ [\![A]\!]_\rho\ [\![\lambda x.B]\!]_\rho)$ and $[\![M]\!]_\rho \in El([\![A]\!]_\rho)$. It follows that we have

$$[\![N\ M]\!]_\rho = [\![N]\!]_\rho\ [\![M]\!]_\rho \in El([\![\lambda x.B]\!]_\rho\ [\![M]\!]_\rho)$$

Since $El([\![A]\!]_\rho) \in \mathsf{TP}(\mathsf{D})$ we have $[\![M]\!]_\rho \neq \bot$. Hence by Theorem 2.16 and Corollary 2.17 we have

$$[\![\lambda x.B]\!]_\rho\ [\![M]\!]_\rho = [\![B]\!]_{\rho, x=[\![M]\!]_\rho} = [\![B[M]]\!]_\rho$$

and so $[\![N\ M]\!]_\rho \in El([\![B[M]]\!]_\rho)$ as expected. □

4.2. **Construction of a model.**

Theorem 4.4. The filter model D of UPL can be extended to a model $T \in \mathsf{TP}(\mathsf{D})$, $El : T \to \mathsf{TP}(\mathsf{D})$.

Proof. The main idea is to define the pair T, El in two inductive steps, using Lemma 2.15 to ensure the consistency of this definition. We define first T_0, El. We have $\top \in T_0$ and $\top \in El(A)$ if $A \in T_0$. Furthermore, we have

- $\mathsf{N}_0 \in T_0$
- $\mathsf{N}_1 \in T_0$ and $\mathsf{0} \in El(\mathsf{N}_1)$
- $\mathsf{Nat} \in T_0$ and $\mathsf{0} \in El(\mathsf{Nat})$ and $\mathsf{S}\ x \in El(\mathsf{Nat})$ if $x \in El(\mathsf{Nat})$
- $A + B \in T_0$ if $A, B \in T_0$ and $\mathsf{Inl}\ x \in El(A + B)$ if $x \in El(A)$ and $\mathsf{Inr}\ y \in El(A + B)$ if $y \in El(B)$
- $A \times B \in T_0$ if $A, B \in T_0$ and $(x, y) \in El(A \times B)$ if $x \in El(A)$ and $y \in El(B)$
- $\mathsf{Fun}\ A\ F \in T_0$ if $A \in T_0$ and $F\ x \in T_0$ for $x \in El(A)$. Furthermore $w \in El(\mathsf{Fun}\ A\ F)$ if $w\ x \in El(F\ x)$ whenever $x \in El(A)$

We can then define $T \supseteq T_0$ and the extension $El : T \to \mathsf{TP}(\mathsf{D})$ by the same conditions extended by one clause

- $\mathsf{N}_0 \in T$
- $\mathsf{N}_1 \in T$ and $\mathsf{0} \in El(\mathsf{N}_1)$
- $\mathsf{Nat} \in T$ and $\mathsf{0} \in El(\mathsf{Nat})$ and $\mathsf{S}\ x \in El(\mathsf{Nat})$ if $x \in El(\mathsf{Nat})$
- $A + B \in T$ if $A, B \in T$ and $\mathsf{Inl}\ x \in El(A + B)$ if $x \in El(A)$ and $\mathsf{Inr}\ y \in El(A + B)$ if $y \in El(B)$
- $A \times B \in T$ if $A, B \in T$ and $(x, y) \in El(A \times B)$ if $x \in El(A)$ and $y \in El(B)$
- $\mathsf{Fun}\ A\ F \in T$ if $A \in T$ and $F\ x \in T$ for $x \in El(A)$. Furthermore $w \in El(\mathsf{Fun}\ A\ F)$ if $w\ x \in El(F\ x)$ whenever $x \in El(A)$
- $\mathsf{U} \in T$ and $El(\mathsf{U}) = T_0$

The definition of the pair T, El is a typical example of an *inductive-recursive* definition: we define simulatenously the subset T *and* the function El on this subset. The justification of such a definition is subtle, but it is standard [2, 8, 22]. It can be checked by induction that $T \in \mathsf{TP}(\mathsf{D})$ and $El(A) \in \mathsf{TP}(\mathsf{D})$ if $A \in T$. The next subsection proves that $[\![h]\!] \in El\ ([\![A]\!])$ if $\vdash h{:}A$ is a typing rule for a constant h. □

4.3. **Strong normalisation via totality.** It is rather straightforward to check that we have $[\![h]\!] \in El([\![A]\!])$ for all the constants $h : A$ that we have introduced except the last two constants Φ and Ψ. For instance $[\![\mathsf{exit}]\!] \in El(\mathsf{N}_0 \to A)$ for any $A \in T$ since $El(\mathsf{N}_0) = \{\top\}$ and $[\![\mathsf{exit}]\!]\ \top = \top$ is in $El(A)$. To check $[\![h]\!] \in El([\![A]\!])$ is more complex for the last two functions.

Theorem 4.5. For all constants $h : A$ that we have introduced, we have $[\![h]\!] \in El([\![A]\!])$.

Proof. To simplify the notations we write h instead of $[\![h]\!]$ if h is a constant of the language, and we say simply that h is total instead of $h \in El(A)$. The only difficult cases are for the constants Φ and Ψ. It is the only place where we use classical reasoning. We only write the proof for Φ, the case of Ψ is similar.

Assume that Φ is not total. We can then find total elements $B \in El(\mathsf{Nat} \to \mathsf{U})$, $H \in El(\mathsf{Fun}\ \mathsf{Nat}\ (\lambda x.\neg\neg\ (B\ x)))$, $K \in El(\neg\ (\mathsf{Fun}\ \mathsf{Nat}\ B))$, $n \in El(\mathsf{Nat})$ and $v \in El(B\ n)$ such that $\Phi\ B\ H\ K\ n\ v$ does not belong to $El(\mathsf{N}_0) = \{\top\}$. Since

$$\Phi\ B\ H\ K\ n\ v = K\ (\lambda x.\Psi\ B\ H\ K\ n\ v\ x\ (\mathsf{less}\ x\ n))$$

and K is total, there exists $x \in El(\mathsf{Nat})$ such that $\Psi\ B\ H\ K\ n\ v\ x\ (\mathsf{less}\ x\ n)$ is not total at type $B\ x$. Given the definition of Ψ this implies that $\mathsf{less}\ x\ n$ is of the form $\mathsf{Inr}\ h$. It follows from the definition of less that n is of the form $\overline{p}$. Furthermore

$$\Psi\ B\ H\ K\ n\ v\ x\ (\mathsf{less}\ x\ n) = \mathsf{exit}\ (H\ \overline{p}\ (\lambda y.\Phi\ H\ K\ \overline{p+1}\ (v, y)))$$

is not total. Since H is total, there exists $y_p \in El\ (B\ \overline{p})$ such that $\Phi\ B\ H\ K\ \overline{p+1}\ (v, y_p)$ is not total. Reasoning in the same way, we see that there exists $y_{p+1} \in El\ (B\ \overline{p+1})$ such that $\Phi\ B\ H\ K\ \overline{p+2}\ (v, y_p, y_{p+1})$ is not total. Thus we build a sequence of elements $y_m \in El\ (B\ \overline{m})$ for $m \geq p$ such that, for any m

$$\Phi\ B\ H\ K\ \overline{m}\ (v, y_p, \ldots, y_{m-1}) \neq \top$$

Consider now an element $x = \overline{q}$. For $m > q$ we have $\mathsf{S}\ x \leq \overline{m} = \mathsf{N}_1$ and we take $f\ x$ to be $\mathsf{get}\ \overline{m}\ x\ 0\ (v, y_p, \ldots, y_{m-1})$. This is well defined since we have for $m_1, m_2 > q$ by Lemma 3.1

$$\mathsf{get}\ B\ \overline{m_1}\ x\ 0\ (v, y_p, \ldots, y_{m_1-1}) = \mathsf{get}\ B\ \overline{m_2}\ x\ 0\ (v, y_p, \ldots, y_{m_2-1})$$

We take also $f\ (\mathsf{S}^q\ \top) = \top$. This defines a total element f in $El\ (\mathsf{Fun}\ \mathsf{Nat}\ (\lambda x.El\ (B\ x)))$. Since K is total, $K\ f$ is total and belongs to $El\ (\mathsf{N}_0) = \{\top\}$. Hence $K\ f = \top$. Since $\top$ is a finite element of D we have by continuity $K\ f_0 = \top$ for some finite approximation f_0 of f. In particular there exists m such that if $g_m\ (\mathsf{S}^q\ 0) = f\ (\mathsf{S}^q\ 0)$ and $g_m\ (\mathsf{S}^q\ \top) = f\ (\mathsf{S}^q\ \top)$, for all $q < m$, then $K\ g_m = \top$. If we define

$$g_m\ x = \Psi\ B\ H\ K\ \overline{m}\ (v, y_p, \ldots, y_{m-1})\ x\ (\mathsf{less}\ x\ \overline{m})$$

we do have $g_m\ (\mathsf{S}^q\ 0) = f\ (\mathsf{S}^q\ 0)$ and $g_m\ (\mathsf{S}^q\ \top) = f\ (\mathsf{S}^q\ \top)$ for all $q < m$. Hence $K\ g_m = \top$. But then

$$\Phi\ B\ H\ K\ \overline{m}\ (v, y_p, \ldots, y_{m-1}) = K\ g_m = \top$$

which contradicts the fact that the element $\Phi\ B\ H\ K\ \overline{m}\ (v, y_p, \ldots, y_{m-1})$ is *not* total. □

Like in [11], it is crucial for this argument that we are using a domain model. These constants make also the system proof-theoretically strong, at least the strength of second-order arithmetic.

Corollary 4.6. If $\vdash A$ then $[\![A]\!] \neq \bot$. If $\vdash M : A$ then $[\![M]\!] \neq \bot$.

Proof. If $\vdash A$ we have by Theorem 4.3 that $[\![A]\!] \in T$. By Theorem 4.4 we have $T \in \mathsf{TP}(\mathsf{D})$. Hence $[\![A]\!] \neq \bot$. Similarly, if $\vdash M : A$ we have by Theorem 4.3 that $[\![A]\!] \in T$ and $[\![M]\!] \in El([\![A]\!])$. By Theorem 4.4 we have $T \in \mathsf{TP}(\mathsf{D})$ and $El([\![A]\!]) \in \mathsf{TP}(\mathsf{D})$. Hence $[\![A]\!] \neq \bot$ and $[\![M]\!] \neq \bot$. □

By combining Corollary 4.6 with Theorem 2.14 we get

Theorem 4.7. If $\vdash A$ then A is strongly normalisable. If $\vdash M : A$ then M is strongly normalisable.

Conclusion

We have built a filter model D for an untyped calculus having the property that a term is strongly normalisable whenever its semantics is $\neq \bot$, and then used this to give various *modular* proofs of strong normalization. While each part uses essentially variation on standard materials, our use of filter models seems to be new and can be seen as an application of computing science to proof theory. It is interesting that we are naturally lead in this way to consider a domain with a top element. We have shown on some examples that this can be used to prove strong normalisation theorem in a modular way, essentially by reducing this problem to show the soundness of a semantics over the domain D. There should be no problem to use our model to give a simple normalisation proof of system F extended with bar recursion. It is indeed direct that totality predicates are closed under arbitrary non empty intersections. By working in the D-set model over D [24, 4], one should be able to get also strong normalisation theorems for various impredicative type theories extended with bar recursion.

For proving normalisation for *predicative* type systems, the use of the model D is proof-theoretically too strong: the totality predicates are sets of filters, that are themselves sets of formal neighbourhoods, and so are essentially third-order objects. For applications not involving strong schemas like bar recursion, it is possible however to work instead only with the definable elements of the set D, and the totality predicates become second-order objects, as usual. It is then natural to extend our programming language with an extra element $\top$

that plays the role of a top-level error. As suggested also to us by Andreas Abel, it seems likely that Theorem 2.11 has a purely combinatorial proof, similar in complexity to the one for simply typed λ-calculus. He gave such a proof for a reasonable subsystem in [1].

A natural extension of this work would be also to state and prove a *density* theorem for our denotational semantics, following [13]. The first step would be to define when a formal neighbourhood is of a given type.

In [6, 18], for untyped λ-calculus without constants, it is proved that a term M is strongly normalizing if *and only if* $[\![M]\!] \neq \bot$. This does not hold here since we have for instance $\mathsf{0}\ \mathsf{Nat}$ strongly normalizing, but $[\![\mathsf{0}\ \mathsf{Nat}]\!] = \bot$. However, it may be possible to find a natural subset of terms M for which the equivalence between M is strongly normalizing and $[\![M]\!] \neq \bot$ holds. Additionally, Colin Riba showed this result for a system where the neighbourhoods are closed by union but were the rewrite rules are weaker [20].

Most of our results hold without the hypotheses that the rewrite rules are mutually disjoint. We only have to change the typing rules for a constant f in Figure 2 by the uniform rule: $\Gamma \vdash_{\mathcal{M}} f : U_1 \to \ldots \to U_k \to V$ if *for all* rules $f\ p_1 \ldots p_k = M$ and *for all* $W_1, \ldots, W_n$ such that $p_i(W_1, \ldots, W_n) = U_i$ we have $\Gamma, x_1 : W_1, \ldots, x_n : W_n \vdash_{\mathcal{M}} M : V$. (This holds for instance trivially in the special case where no rules for f matches $U_1, \ldots, U_n$.) For instance, we can add a constant $+$ with rewrite rules

$$\begin{array}{lcccl} + & n & \mathsf{0} & = & n \\ + & \mathsf{0} & n & = & n \\ + & n & (\mathsf{S}\ m) & = & \mathsf{S}\ (+\ n\ m) \\ + & (\mathsf{S}\ n) & m & = & \mathsf{S}\ (+\ n\ m) \end{array}$$

and Theorem 2.14 is still valid for this extension.

Acknowledgement

Thanks to Mariangiola Dezani-Ciancaglini for the reference to the paper [6]. The first author wants also to thank Thomas Ehrhard for reminding him about proofs of strong normalisation via intersection types.

References

[1] A. Abel. Syntactical Normalization for Intersection Types with Term Rewriting Rules. 4th International Workshop on Higher-Order Rewriting, HOR'07, Paris, France, 2007.

[2] P. Aczel. Frege structures and the notions of proposition, truth and set. *The Kleene Symposium*, pp. 31–59, Stud. Logic Foundations Math., 101, North-Holland, Amsterdam-New York, 1980.

[3] Y. Akama. SN Combinators and Partial Combinatory Algebras. LNCS 1379, p. 302-317, 1998.

[4] Th. Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, 1993.

[5] R. Amadio and P.L. Curien. *Domains and Lambda-Calculi.* Cambridge tracts in theoretical computer science, 46, (1997).

[6] S. van Bakel. Complete restrictions of the Intersection Type Discipline. Theoretical Computer Science, 102:135-163, 1992.

[7] H. Barendregt, M. Coppo and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic* 48 (1983), no. 4, 931–940 (1984).

[8] M. Beeson. *Foundations of constructive mathematics. Metamathematical studies.* Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)], 6. Springer-Verlag, Berlin, 1985.

[9] S. Berardi, M. Bezem and Th. Coquand. On the computational content of the axiom of choice. Journal of Symbolic Logic 63 (2), 600-622, 1998.
[10] U. Berger and P. Oliva. Modified Bar Recursion and Classical Dependent Choice. Logic Colloquium '01, 89–107, Lect. Notes Log., 20, Assoc. Symbol. Logic, Urbana, IL, 2005.
[11] U. Berger. Continuous Semantics for Strong Normalisation. LNCS 3526, 23-34, 2005.
[12] U. Berger. Strong normalization for applied lambda calculi. Logical Methods in Computer Science, 1-14, 2005.
[13] U. Berger. Continuous Functionals of Dependent and Transfinite Types. in *Models and Computability*, London Mathematical Society, Lecture Note Series, p. 1–22, 1999.
[14] J. W. Klop, V. van Oostrom and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. Theoretical Computer Science, Volume 121, No. 1 & 2, pp. 279 - 308, December 1993.
[15] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In *Constructivity in Mathematics*, North-Holland, 1958.
[16] P. Martin-Löf. Lecture note on the domain interpretation of type theory. *Workshop on Semantics of Programming Languages, Chalmers*, (1983).
[17] P.Martin-Löf. An intuitionistic theory of types. in *Twenty-five years of constructive type theory* (Venice, 1995), 127–172, Oxford Logic Guides, 36, Oxford Univ. Press, New York, 1998.
[18] G. Pottinger. A type assignment for the strongly normalizable terms. in: J.P. Seldin and J.R. Hindley (eds.), *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, Academic Press, London, pp. 561-577, 1980.
[19] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223-255, 1977.
[20] C. Riba. Strong Normalization as Safe Interaction. Logic In Computer Science 2007.
[21] D. Scott. Lectures on a mathematical theory of computation. Theoretical foundations of programming methodology (Munich, 1981), 145–292, NATO Adv. Study Inst. Ser. C: Math. Phys. Sci., 91, Reidel, Dordrecht, 1982.
[22] D. Scott. Combinators and classes. *λ-calculus and computer science theory*, pp. 1–26. Lecture Notes in Comput. Sci., Vol. 37, Springer, Berlin, 1975.
[23] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In F.D.E.Dekker, editor, Recursive Function Theory, 1962
[24] Th. Streicher. *Semantics of Type Theory.* in the series Progress in Theoretical Computer Science. Basel: Birkhaeuser. XII, 1991.
[25] W.W. Tait. Normal form theorem for bar recursive functions of finite type. *Proceedings of the Second Scandinavian Logic Symposium*, North-Holland, 1971.