Termination with Domain Theory

Jonas Oberhauser
Saarland University

September 23, 2014

Abstract

Typed lambda calculi can be used as computational logical systems. Logical consistency of some type theories can be reduced to the property of Termination, but Termination proofs often require new technology for each system. We study and formally verify a Termination proof framework by Bernadet and Lengrand that promises straightforward proofs for Termination up to System $\mathcal{F}$.

Contents

Chapter 1

Introduction

It is well known that through the Curry-Howard-de Bruijn isomorphism, typed lambda calculi can be seen as logical systems. A system is said to be terminating if all reduction paths yield a normal form that can not be reduced further. In many popular variants, this property implies logical consistency. However, Termination proofs often require complex techniques, which have to be repeated and expanded for each new complex system ([3], [10], [6], [7], [8]).

In 2006, Coquand and Spiwack [4] first suggested to use a domain theoretic framework for proving Termination, which reduces this complexity considerably. This framework consists of a terminating target system, which is used to define a model for the terminating fragment of untyped lambda calculus (we thus consider it a premodel of untyped lambda calculus). In order to prove Termination of any typing system for lambda calculus, it suffices to show that the model of its terms is non-empty. The last step is very straightforward and does not require new ideas even for, e.g., dependent type theories. Sadly we see no way to formalize the full framework in Coq due to the usage of induction-recursion schemes.

We therefore turned to a 2012 paper by Bernadet and Lengrand [2], who developed a similar framework. While this framework is less versatile and can show Termination only for systems up to System $\mathcal{F}$, it is also much easier to formalize. With only minor modifications to this system, we could formalize Termination proofs of four logical systems: STLC, CL, System $\mathcal{F}$, and System $\mathcal{T}$. The proofs turn out to be very short and straightforward, as Table 1 suggests.

This document describes the system and our formalization of it in more detail. Furthermore we try to point out places where we now believe improvements could be made in the development, either to match the mathematical speech more precisely or simply shorten the formalization. In Chapter 2 we describe our target system. The premodel is defined in Chapter 3. We give an overview of the systems that we covered in Chapter 4 along with the definitions required to make the proofs go through. Finally, we give a short overview of the possible shortcomings of the framework and consider Coquand's framework in Chapter 5.

System	Without the Framework	With the Framework
Simply Typed LC	124	37
Combinatory Logic	ca. 200[1]	89
System $\mathcal{F}$	249[2]	105
System $\mathcal{T}$	–	208

Table 1.1: Length of Termination proofs in Coq of various systems (according to coqwc). All proofs use Autosubst[9] if applicable. We could not find and did not attempt a suitable formal proof without the framework for System $\mathcal{T}$, due to the fact that Termination proofs for System $\mathcal{T}$ are generally considered too tedious to formalize.

Chapter 2

The Target System

2.1 Definitions.v: Underlying Definitions

This file contains the definitions related to the target system, bar reduction.

We first define the syntax of untyped lambda calculus, ULC. Note that we use the Autosubst library (and thus a de Bruijn representation[5]), which comes with all necessary laws concerning binders and substitutions. Autosubst also provides the `bind` function, which we use for defining our term syntax:

```
Inductive term :=
| TVar (x : var)
| App (s t : term)
| Lam (s : {bind term}).
```

In order for coq to accept `(Lam 0) (Lam 0)` as a term $((\lambda x.x)\ (\lambda x.x))$, we define some coercions:

```
Coercion App : term >-> Funclass.
Coercion TVar : var >-> term.
```

The definition for the types of the system is in three levels, two of which are mutually inductive.

$$F, G, H ::= \alpha \mid A \to F$$
$$A, B, C ::= A \circ B \mid F$$

$$U, V, W ::= \omega \mid A$$

As in Bernadet [2], we call these levels F-Types, A-Types, and U-Types, respectively. Here are some thoughts on the individual classes:

F-Types: These are either variables or arrow types. Note that the domains of the arrow can be a binary tree of F-Types, while the codomain is again an F-Type.

A-Types: A-Types are binary trees of F-Types. We call the constructor concatenation. If types in the system are seen as resources, then $A \circ B$ are the resources in A and the resources B.

U-Types: These can be seen as option types, where ω corresponds to the empty constructor (and thus to having no resources). Note that U-Types can not occur inside of A-Types, e.g., as $\alpha \to \omega \to \alpha$. If U-Types could occur in those places, the type theory would correspond to normalization[1], not Termination.

We will later abstract away from the tree structure of A-Types, and it is possible to define A-Types as nonempty lists of F-Types. The current definition is historical and, suprisingly, seems to simplify some of the proofs. There are a few other options, e.g., treating U-Types as lists of F-Types, where the list being empty corresponds to ω, and the list being nonempty corresponds to it being an A-Type. However, such representations do not seem to improve the complexity of the formalization.

This is why the formalization follows the representation given above, except for the names which have been changed to `basetype`, `inttype`, and `type`:

```
Inductive basetype :=
| BASE (x : nat)
| Arr (A : inttype) (F : basetype)
with inttype :=
| UpB (F : basetype)
| Int (A B : inttype).
Inductive type :=
| UpI (A : inttype)
| OMEGA.

Infix "-->" := Arr.
Notation "A 'o' B" := (Int A B) (at level 50).
```

The concatenation constructor can be lifted to the concatenation operator on U-Types, which for U-Types is defined as follows:

$$\omega \circ U = U$$
$$U \circ \omega = U$$

This definition is captured by `int` in the code:

[1]The property that every term has a reduction path that yields a normal form, as opposed to all reduction paths yielding a normal form.

```
Definition int U V : type :=
  match U, V with
  | OMEGA, V => V
  | U, OMEGA => U
  | UpI A, UpI B => (Int A B)
  end.

Infix "*" := int.
```

As promised before, we abstract away from the tree structure of A-Types. We consider two U-Types equivalent if they contain the exact same resources; e.g., $A \circ (B \circ C) \equiv C \circ (B \circ A)$, but $A \circ A \not\equiv A$[2]. A more formal definition can be given by rules:

$$\frac{}{A \circ B \equiv B \circ A} \qquad \frac{}{A \circ (B \circ C) \equiv (A \circ B) \circ C} \qquad \frac{}{(A \circ B) \circ C \equiv A \circ (B \circ C)}$$

$$\frac{}{F \equiv F} \qquad \frac{}{\omega \equiv \omega} \qquad \frac{A \equiv A' \qquad B \equiv B'}{A \circ B \equiv A' \circ B'} \qquad \frac{A \equiv B \qquad B \equiv C}{A \equiv C}$$

The formalization of this definition is straightforward[3] :

```
Inductive type_equiv : type -> type -> Prop :=
| te_refl_F F :
  type_equiv F F
| te_refl_OMEGA :
  type_equiv OMEGA OMEGA
| te_comm_Int A B :
  type_equiv (A o B) (B o A)
| te_cong_Int A B A' B' : type_equiv A A' -> type_equiv B B'
  -> type_equiv (A o B) (A' o B')
| te_trans_ABC A B C : type_equiv A B -> type_equiv B C ->
  type_equiv A C
| te_assoc_Int_lr A B C :
  type_equiv ((A o B) o C) (A o (B o C))
| te_assoc_Int_rl A B C :
  type_equiv (A o (B o C)) ((A o B) o C)

Infix "==" := type_equiv (at level 60) : type_scope.
```

[2]Since we consider U-Types only up to equivalence, it sounds plausible to drop parentheses and concatenation symbols in all places. We do not do this.

[3]We will often see statements involving equivalence on A-Types. Their proofs are cumbersome in Coq. This is due to induction being tedious to set up. Furthermore, this leads to equality between A-Types being cast up to an equality between U-Types, which makes, e.g., `subst` fail. Consider as an example the prelude of `equiv_closure` in Filters.v (which is too large to show here).

A definition like the following could be used to simplify those proofs tremendously:

```
Definition extr U := match U with OMEGA => BASE 0 | UpI A => A end.
```

Alternatively, type equivalence could be defined as a relation on A-Types, with a second definition covering the case where both types are ω.

We use this to define subsumption, which is a preorder on U-Types that captures the intuition of having more resources:

$$U \subseteq V := \exists U'.U \circ U' \equiv V$$

In the formalization the order is swapped, the symbolic notation reflects this:

```
Definition subsumes U V := exists V', U == V * V'.

Infix ">=" := subsumes.
```

Contexts Γ, Δ, E are functions that map variables to U-Types. Another perspective is that they are partial functions mapping variables to A-Types. The empty context $\emptyset$ maps all variables to ω. The concatenation operator, equivalence, and subsumption are each lifted pointwise to contexts.

In the formalization, subsumption is not lifted pointwise. Doing so seems to require a fact that is tedious to formalize, namely that we can construct the witness in the case of subsumption.

```
forall U V, (exists U' , U o U' == V) -> { U' | U o U' == V }
```

While this is conceivable - an algorithm deleting `U` from `V` should do the trick - it also seems like a huge effort, especially with the tree-based representation of A-Types.

Therefore subsumption on contexts is defined as follows:

$$\Gamma \subseteq \Delta \iff \exists \Gamma'.\Gamma \circ \Gamma' \equiv \Delta$$

Note again that the subsumption has swapped the order:

```
Definition Empty x := OMEGA.

Definition cint Gamma Delta x := Gamma x * Delta x.
Arguments  cint _ _ _ /.

Infix "&" := cint (at level 50).


Definition PWR {X Y} (R : Y -> Y -> Prop)
  (f g : X -> Y) := forall x : X, R (f x) (g x).
Infix "=1" := (PWR eq)  (at level 60).
Definition cte := PWR (X := var) type_equiv.
Infix "==1" := cte (at level 60).
Definition csub Gamma Delta := exists Delta', Gamma ==1 Delta & Delta'.
Infix ">=1" := csub (at level 70).
```

For the typing judgement, we started out with the system from Bernadet and Lengrand, except for the variable and ω rules. Those we changed as follows:

$$\frac{\Gamma\ x = F}{\Gamma \vdash_0 x : F} \qquad\qquad \frac{\Gamma = (x \mapsto \omega)}{\Gamma \vdash_0 t : \omega}$$

I.e., we dropped the requirement that unused variables do not occur in the context. This was in the hope that it would simplify the system. It turns out that the simplification is minimal, and it seems that in order for the Bernadet and Lengrand proof to go through in Coq, the extraction of the witness is needed (as mentioned above).

Luckily, the type theory can be simplified further by allowing for three key properties:

1. Monotonicity under subsumption on contexts: if $\Gamma \subseteq \Gamma'$ then $\Gamma \vdash_n t : U$ implies $\Gamma' \vdash_n t : U$
2. Monotonicity in the cost: if $n < m$, then $\Gamma \vdash_n t : U$ implies $\Gamma \vdash_m t : U$
3. Anti-monotonicity in the type: $\Gamma \vdash_n t : U$ then for all $V \subseteq U$, $\Gamma \vdash_n t : V$

A system exposing these properties is the following, which is also used in the formalization:

$$\frac{F \subseteq \Gamma\ x \qquad n \in \mathbb{N}}{\Gamma \vdash_n x : F} \qquad \frac{\Gamma \circ \Delta \equiv E \qquad \Gamma \vdash_n s : A \to F \qquad \Delta \vdash_m t : A}{E \vdash_{n+m+1} s\ t : F}$$

$$\frac{\Gamma \circ \Delta \equiv E \qquad \Gamma \vdash_n t : A \qquad \Delta \vdash_m t : B}{E \vdash_{n+m} t : A \circ B} \qquad \frac{\Gamma, x : A \vdash_n s : F}{\Gamma \vdash_n \lambda x.s : A \to F} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash_n s : \omega}$$

The following Coq definition matches this perfectly:

```
Inductive ty : nat -> context -> term -> type -> Prop :=
| tyVar Gamma n x F : Gamma x >= F ->
  ty n Gamma x F
| tyAbs Gamma n s A F : ty n ((A : type) .: Gamma) s F ->
  ty n Gamma (Lam s) (A --> F)
| tyApp Gamma n m Delta s t A F :
  ty n Gamma s (A --> F) ->
  ty m Delta t A ->
  forall Epsilon (e : Gamma & Delta ==1 Epsilon),
    ty (S (n + m)) Epsilon (s t) F
| tyInt Gamma n A Delta m B s :
  ty n Gamma s A ->
  ty m Delta s B ->
  forall Epsilon (e : Gamma & Delta ==1 Epsilon),
    ty (n+m) Epsilon s (A o B)
| tyO Gamma n s :
  ty n Gamma s OMEGA.
```

As a direct consequence, the basic properties (Lemma 5, p. 8) change; for more details, see 2.3.

For those not familiar with the original system, there are a few noteworthy facts. First, note the index of the turnstyle. It is a particular fact of this system that this index is an upper bound on the number of reduction steps a term can make. This is becomes clear from three observations:

- Every application increases this bound by one, as apparent from the application rule.
- Every occurence of a variable - and thus every time a term is duplicated due to a beta reduction - requires one additional F-Type in the context. This is also a consequence of the application rule, where the context is split, i.e., every F-Type is assigned either to the left or the right side of the application.
- Each F-Type that a term is given increases its cost by the number of reduction steps this term can make, as apparent from the concatenation rule.

Thus, by typing a term multiple times with the concatenation rule, we pay in advance for all reduction steps its duplications can do after a beta reduction.

Second, note that while ω can be assigned as a type for every term, this rule can only be used as the last rule in a derivation.

Third, this system types exactly the terminating terms. We did not consider a proof of this claim; however, one can find such a proof in Bernadet [2]. As an example of a terminating term which in typical type systems does not have a type consider $\lambda x.x\ x$:

$$\Gamma \vdash_1 \lambda x.x\ x : \alpha \circ (\alpha \to \beta) \to \beta$$

Fourth, a form of substitutivity holds on the types and contexts: if $\Gamma \vdash t : U$, then $\sigma\ \Gamma \vdash t : \sigma\ U$ if σ can be applied to Γ and U[4]. We have not formalized a proof of this fact. However, we make use of it in our Termination proof for System $\mathcal{T}$.

This concludes our description of Definitions.v.

2.2 Equivalence.v, Subsumption.v, Contexts.v: Properties of these Relations and Their Lifting to Contexts

These files contain proofs of the properties, e.g., that equivalence is an equivalence relation, subsumption is a preorder, etc. There are only few facts that are worthy of mention here.

The first is the proof of Lemma 2.5 (p. 6), which is that adding resources breaks type equivalence[5]:

$$U \circ V \equiv U \implies V = \omega$$

This lemma is formalized as follows:

[4] As a counterexample, consider $\sigma^{\alpha,\beta}_{\alpha,(\beta\circ\beta)}$ which can be applied to $\beta \to \alpha$ but not to $\alpha \to \beta$, since $\alpha \to \beta \circ \beta$ is not a type.

[5] The following equivalent statement makes this clear: $U \circ A \not\equiv U$

```
Lemma te_inv U V : U * V == U -> V = OMEGA.
```

Here, the proof has changed. The new proof goes as follows: we first show that the number of concatenations is stable under equivalence. However, if V is not ω, $U \circ V$ has more concatenations than U. This proves the claim. We consider this a minor simplification over the original proof.

Another fact is that if an F-Type is contained in a concatenation, it is contained in one of the two original types:

$$F \subseteq A \circ B \to F \subseteq A \vee F \subseteq B$$

While this is obvious, due to the current definition of subsumption, the constructive proof is not straightforward. We first restrict subsumption to F-Types. A formalization of this is as follows:

```
Inductive includes : type -> basetype -> Prop :=
| includesDropL A B F : includes B F -> includes (A o B) F
| includesDropR A B F : includes A F -> includes (A o B) F
| includesF F : includes F F
.

Lemma incl_inv_sub {U F} : includes U F -> U >= F.
```

This property is stable under equivalence:

```
Lemma te_inv_incl {U V} : U == V ->
  forall F, includes V F -> includes U F.
```

This allows us to show the original claim. Since $F \circ U \equiv A \circ B$, and F is included in $F \circ U$, it is also included in $A \circ B$. A simple inversion on this fact concludes the proof. A formalization can be found in the following lemma:

```
Lemma sub_inv_int {A B F} : A o B >= F -> A >= F \/ B >= F.
```

Adding to our earlier point about subsumption on contexts, the following property is of interest:

```
Lemma csub_sub {Gamma Delta} : Gamma >=1 Delta ->
  forall x, Gamma x >= Delta x.
```

Considering the fact that the former can be shown in all circumstances that show up in the proof, there is no difference between the two definitions in practice.

This concludes the interesting facts in these files.

2.3 Types.v: Properties of the Typing Relation

The file starts with generalizations of the inversion properties. There are two of them: inversion of variables and inversion of concatenation.

1. When $\Gamma \vdash_n x : A$, then $A \subseteq \Gamma\ x$.

2. When $\Gamma \vdash_n t : U \circ V$, then there are Γ_1, Γ_2, n_1, and n_2, such that $n = n_1 + n_2$, $\Gamma \equiv \Gamma_1 \circ \Gamma_2$, $\Gamma_1 \vdash_{n_1} t : U$, and $\Gamma_2 \vdash_{n_2} t : V$.

Note that the variable rule holds for F-Types and the concatenation rule for A-Types. The rules above are gnerealizations to A-Types and U-Types, respectively. The two properties are formalized as follows:

```
Lemma invtyVar n Gamma x A : ty n Gamma x A ->
  Gamma x >= A.

Lemma invtyInt {n Gamma t U V} : ty n Gamma t (U * V) ->
  exists n1 n2 Gamma1 Gamma2,
    n = n1 + n2 /\
    Gamma1 & Gamma2 ==1 Gamma /\
    ty n1 Gamma1 t U /\
    ty n2 Gamma2 t V.
```

Furthermore, a generalized variable rule is admissible: if $U \subseteq \Gamma x$, then $\Gamma \vdash_n x : U$ for all $n \in \mathbb{N}$. This fact is formalized in `tyVarU`:

```
Lemma tyVarU n Gamma x U :
  Gamma x >= U ->
  ty n Gamma x U.
```

Recall the properties mentioned in Section 2.1, which are proven here:

1. When $\Gamma \subseteq \Gamma'$, then $\Gamma \vdash_n t : U$ implies $\Gamma' \vdash_n t : U$.
2. If $n < m$, then $\Gamma \vdash_n t : U$ implies $\Gamma \vdash_m t : U$
3. When $\Gamma \vdash_n t : U$ and $V \subseteq U$, then $\Gamma \vdash_n t : V$.

The latter is a well known property from intersection type systems, e.g., Barendregt [1], and is often called the subtyping rule. We formalized a slightly less general version, where $U, V \neq \omega$.

It is noteworthy that its proof is by its two main special cases, namely that $\Gamma \vdash_n t : A \circ B$ implies $\Gamma \vdash_n t : A$ and that $U \equiv V$ implies $\Gamma \vdash_n t : U \iff \Gamma \vdash_n t : V$.

```
Lemma tycsub {n Gamma t U} :
  ty n Gamma t U ->
  forall Gamma', Gamma' >=1 Gamma ->
    ty n Gamma' t U.

Lemma tymon {n Gamma t U} :
  ty n Gamma t U ->
  forall m, n <= m ->
    ty m Gamma t U.

Lemma equi_type {U V} :
  U == V -> forall {n Gamma t},
```

```
    ty n Gamma t U -> ty n Gamma t V.
Lemma tysubA {A B} :
  A >= B -> forall {n : nat} {Gamma t},
    ty n Gamma t A -> ty n Gamma t B.
```

2.4 Weakening.v and Substitution.v: Proving the Typing of Implicit Substitutions

Since we committed ourselves to a de Bruijn representation, we proved the typing of implicit substitutions (Lemma 8, p. 9) with *parallel substitutions* rather than single variable substitutions.

However, only a subset of substitutions have the properties required for the proof to go through. We define one such class inductively: we say σ translates from Γ to Δ with cost n and write $\sigma \vdash_n \Gamma \to \Delta$.

$$\frac{\text{injective } \xi \qquad \forall x.\Gamma\ x = \Delta\ (\xi\ x)}{\xi \vdash_0 \Gamma \to \Delta}$$

$$\frac{\sigma \vdash_n \Gamma \to \Delta \qquad \Delta' \vdash_m t : U \qquad E \equiv \Delta \circ \Delta'}{\sigma_t^x \vdash_{n+m} \Gamma, x{:}U \to E}$$

In the development, the renaming case is taken care of by the definition `tyren`. Note that injectivity is replaced by a constructive version and that costs are omitted:

```
Inductive tyren xi Gamma Delta : Prop :=
| tyrenInv (ceta : var -> option var) :
    (forall v x, ceta v = Some x -> xi x = v) ->
    (forall x, ceta (xi x) = Some x) ->
    (forall x, Gamma x = Delta (xi x)) ->
      tyren xi Gamma Delta.
```

The general case is handled in the definition `tysub`.

```
Inductive tysub : (var -> term) -> context -> nat -> context -> Prop :=
| tysubRen xi Gamma Delta :
  tyren xi Gamma Delta ->
    tysub xi Gamma 0 Delta
| tysubCons {sigma Gamma n Delta} : tysub sigma Gamma n Delta ->
  forall {Deltat m t U},
    ty m Deltat t U ->
    forall Gamma', Gamma' =1 U .: Gamma ->
    forall Delta', Deltat & Delta ==1 Delta' ->
      tysub (t .: sigma) Gamma' (n + m) Delta'.
```

Then the Substitution Lemma we want to show is as follows: if σ is a translation from Γ to Δ with cost m, $\Gamma \vdash_n t : A$ implies $\Delta \vdash_{n+m} \sigma\ t : A$.

Which in the code is captured by the theorem `tysubst`:

```
Theorem tysubst {n Gamma s A} :
  ty n Gamma s A ->
  forall {sigma Delta m},
    tysub sigma Gamma m Delta ->
    ty (n + m) Delta s.[ sigma ] A.
```

This is a rather tedious theorem in the development, and requires some intermediary steps. First, we need to show that if the source context of a translation can be split, the target context can be split accordingly: if $\sigma \vdash_n \Gamma \to \Delta$ and $\Gamma \equiv \Gamma_1 \circ \Gamma_2$, then there are n_1, n_2, Δ_1, and Δ_2 such that $n = n_1 + n_2$, $\Delta \equiv \Delta_1 \circ \Delta_2$, $\sigma \vdash_{n_1} \Gamma_1 \to \Delta_1$ and $\sigma \vdash_{n_2} \Gamma_2 \to \Delta_2$.

This lemma is distributed over `tyren_split` and `tysub_split`, handling the renaming case and the general case, respectively.

```
Lemma tyren_split {xi Gamma Delta} :
  tyren xi Gamma Delta ->
  forall {Gamma1 Gamma2}, Gamma1 & Gamma2 ==1 Gamma ->
    exists Delta1 Delta2,
      tyren xi Gamma1 Delta1 /\
      tyren xi Gamma2 Delta2 /\
      Delta ==1 Delta1 & Delta2.

Lemma tysub_split {sigma Gamma n Delta} :
  tysub sigma Gamma n Delta ->
  forall {Gamma1 Gamma2}, Gamma1 & Gamma2 ==1 Gamma ->
    exists n1 n2 Delta1 Delta2,
      n = n1 + n2 /\
      Delta ==1 Delta1 & Delta2 /\
      tysub sigma Gamma1 n1 Delta1 /\
      tysub sigma Gamma2 n2 Delta2.
```

As a second step, we have to prove two special cases of the substitution theorem. The first is that the theorem holds for renamings: if ξ is a translation from Γ to Δ (with cost 0), $\Gamma \vdash_n t : A$ implies $\Delta \vdash_n \xi\ t : A$

A consequence of this lemma is known as weakening: if x does not occur free in t, $\Gamma \vdash t : V$ implies $\Gamma, x : U \vdash t : V$

These two lemmas are formalized as `weaken` and `weakening`, respectively:

```
Theorem weaken n Gamma s U :
  ty n Gamma s U ->
  forall xi Delta,
    tyren xi Gamma Delta ->
    ty n Delta s.[ren (xi)] U.

Corollary weakening n Gamma s U :
  ty n Gamma s U ->
  forall V,
    ty n (V .: Gamma) s.[ren (+1)] U.
```

Finally, we have to show that the theorem holds if t is a variable and A is

an F-Type: if σ is a translation from Γ to Δ with cost n, then $F \subseteq \Gamma\ x$ implies $\Delta \vdash_n \sigma\ x : F$.

```
Lemma tysubst_var {Gamma x F} :
  Gamma x >= F ->
  forall sigma Delta n,
    tysub sigma Gamma n Delta ->
    ty n Delta (sigma x) F.
```

2.5 Reduction.v and Termination.v: Termination of the Target System

We first define beta reduction $\rightarrow_\beta$ of ULC. The formalization is straightforward:

```
Inductive step : term -> term -> Prop :=
| step_beta (s t : term) :
    step ((Lam s) t) (s.[t/])
| step_appL (s1 s2 t : term) :
    step s1 s2 -> step (s1 t) (s2 t)
| step_appR (s t1 t2 : term) :
    step t1 t2 -> step (s t1) (s t2)
| step_lam  (s1 s2 : term) :
    step s1 s2 -> step (Lam s1) (Lam s2).
```

The transitive closure of beta reduction $\rightarrow_\beta^+$ is formalized as follows:

```
Inductive steps s t : Prop :=
| one : step s t -> steps s t
| more s' : step s s' -> steps s' t -> steps s t.
```

We define terminating relations inductively:

$$\frac{\forall y, x\ R\ y \rightarrow \mathrm{GSM}_R\ x}{\mathrm{GSM}_R\ y}$$

Termination (in the code `SN`) is thus the property that beta reduction is terminating:

```
Inductive GSN {X : Type} R (x : X) : Prop
  := gsn : (forall y : X, R x y -> GSN R y) -> GSN R x.

Definition SN := GSN step.
```

It is obvious from the application rule that reducible terms have nonzero cost: if s is reducible and $\Gamma \vdash_n s : A$, then $n > 0$.

```
Lemma step_n {n Gamma s A} :
  ty n Gamma s A ->
  forall {s'}, step s s' ->
    n > 0.
```

Furthermore, it is straightforward to show with the Substitution Lemma that reduction reduces the cost: if s reduces to s', then $\Gamma \vdash_n s : A$ implies $\Gamma \vdash_{n-1} s : A$. This theorem is comparable to subject reduction in other systems, but much stronger: it directly yields Termination: if $\Gamma \vdash_n t : A$, then beta reduction terminates on t.

The two are formalized as `step_ty` and `sn_system`, respectively:

```
Lemma step_ty {n Gamma s A} :
  ty n Gamma s A ->
  forall s', step s s' ->
    ty (pred n) Gamma s' A.

Theorem sn_system n Gamma t A :
  ty n Gamma t A -> SN t.
```

Chapter 3

Premodel of ULC

In this chapter we discuss the premodel of untyped lambda calculus. We say premodel because it is a lambda model only for the terminating fragment of ULC (cf., e.g., Lemma 3.5 in Barendregt [1]). The domain of the model are sets of A-Types with two important properties: closure under concatenation and under subsumption. Therefore these sets are called filters in accordance with the set theoretic definition.

3.1 Filter.v: The Domain of the Premodel

Filters are sets of A-Types that are closed under concatenation and subsumption. To be more precise, a set u of A-Types is a filter if:

- $A \in u$ and $B \in u$ imply $A \circ B \in u$, and
- $A \in u$ and $B \subseteq A$ implies $B \in u$

The definition is captured as follows [1]:

```
Record IFilter u := {
 closed_int A B : u A -> u B -> u (A o B) ;
 closed_leu A B : u A -> A >= B -> u B
}.
```

[1] This turns out to be a cumbersome definition which creates many artifacts and hurts automation. A better formalization could be the following (modulo naming):

```
Record IFilter' := {
 u : inttype -> Prop;
 closed_int A B : u A -> u B -> u (A o B) ;
 closed_leu A B : u A -> A >= B -> u B
}.

Coercion u : IFilter' >-> Funclass.
```

Note that `IFilter' : Type`, while `IFilter u : Prop`.

If m is a set of F-Types, then $\langle m \rangle$ is the least filter containing m.

This is formalized with an inductive definition, where `closure m` is $\langle m \rangle$ [2]:

```
Inductive closure m : inttype -> Prop :=
| inF F : m F -> closure m F
| inI A B : closure m A -> closure m B -> closure m (A o B).

Lemma closure_filter m : IFilter (closure m).
```

We can now define application in the model:

$$u@v := \langle \{F \mid \exists A, A \to F \in u \wedge A \in v\} \rangle$$

In Coq, we use an intermediary definition `app`:

```
Inductive app u v F : Prop :=
| inApp A : u (A --> F) -> v A -> app u v F.

Definition int_app u v := closure (app u v).
```

Many theorems hold with equality in mathematics, but only with extensional equality in Coq. Therefore we introduce the symbol `~` for extensional equality.

```
Definition filter_equiv u v := forall A, u A <-> v A.

Infix "~" := filter_equiv (at level 70).
```

We furthermore introduce $\top$ and $\bot$ as the set of all A-Types and the empty set, respectively. Both are filters:

```
Definition top A := True.

Lemma top_filter : IFilter top.

Definition bot A := False.

Lemma bot_filter : IFilter bot.
```

An important property of application is that $\top @ u = \top$ if $u \neq \emptyset$. In the formalization, the property needs to be stated in a constructive way:

```
Lemma top_app {u} : (exists A, u A) -> top ~ top @ u.
```

[2] The following recursive definition would improve automation, but might be considered slightly less intuitive:

```
Fixpoint closure m A : Prop :=
  match A with
  | UpB F => m F
  | A o B => closure m A /\ closure m B
  end
```

3.2 Interpretation.v: The Definition of the Premodel

We start with enviroments ρ, which map variables to filters. Due to our current definition of filters, we have to formalize this in two steps:

```
Definition type_context := var -> inttype -> Prop.

Definition valid rho := forall x, IFilter (rho x).
```

Note that `rho` is an environment only if it is `valid`.

A context is compatible with an environment if pointwise elementship is satisfied. Factoring in that contexts are partial functions, we define:

$$\Gamma \in \rho := \forall x, \Gamma\ x \neq \omega \implies \Gamma\ x \in \rho\ x$$

In the formalization, this will not type and we define instead:

```
Definition compatible rho Gamma := forall x A, Gamma x = A -> rho x A.
```

Now we define the interpretation of a term under ρ as the set of A-Types that can be given under compatible contexts.

$$[t]_\rho := \{A \mid \exists n, \Gamma \in \rho.\ \Gamma \vdash_n t : A\}$$

Recall that the target system terminates. Therefore, if the interpretation is non-empty, the term terminates.

The formalization is straightforward:

```
Inductive int_t rho t A : Prop :=
| intty (n : nat) Gamma :
  compatible rho Gamma -> ty n Gamma t A ->
    int_t rho t A.
```

Due to the concatenation rule and anti-monotonicity of the typing relation in the type, this is clearly a filter. This fact is formalized in `int_IF`:

```
Lemma int_IF rho t : valid rho -> IFilter [t] rho.
```

3.3 Premodel.v: The Interpretation Yields a Premodel

In this file we show that the interpretation yields a premodel. The properties are those of a model, except for the beta substitution, which does not hold in general:

- $[x]_\rho = \rho\ x$

- $[s\ t]_\rho = [s]_\rho @ [t]_\rho$
- $[\lambda x.s]_\rho @ u = [s]_{\rho^x_u}$ if $u \neq \emptyset$

The properties are formalized as follows:

```
Lemma int_x rho x : valid rho ->
  [ x ] rho ~ rho x.

Lemma int_st rho s t : valid rho ->
  [ s t ] rho ~ [ s ] rho @ [ t ] rho.

Lemma int_lam rho s u : valid rho -> IFilter u ->
  (exists B, u B) ->
    [ s ] (u .: rho) ~ [ Lam s ] rho @ u.
```

3.4 RealizorPredicates.v: Realizability Predicates

This file describes realizability predicates and some typical instances. The most common types can be interpreted using only the instances given in this file. However, many inductive types need their own inductively defined realizability predicate; one example can be found in Section 4.3.

A realizability predicate is a set of filters that includes $\top$ and excludes $\bot$

In the formalization, we need to make explicit a few mathematical notions. The exclusion of $\bot$ has to be constructive, and we state that realizability candidates must be closed under extensional equality[3] :

```
Definition realizorCandidate := (inttype -> Prop) -> Prop.
Implicit Types X Y : realizorCandidate.

Record realizorPredicate X := realizor {
  excl_bot u : X u -> (exists A, u A) ;
  incl_top : X top;
  only_filter u : X u -> IFilter u ;
  equiv_closed u v : u ~ v -> X u -> X v
}.
```

[3]The same critique as with `IFilter` applies. Furthermore, the adjective has been mistakenly changed from "realizability" to "realizor". Assuming the definition of `IFilter'` given before, we should thus define:

```
Record realizabilityPredicate :=  {
  X : IFilter' -> Prop;
  excl_bot u : X u -> (exists A, u A) ;
  incl_top : X top;
  equiv_closed u v : u ~ v -> X u -> X v
}.

Coercion X : realizabilityPredicate >-> Funclass.
```

We define the following instances:

- $\{\top\}$ is a realizability predicate
- If X, Y are realizability predicates, so is their functional product $X \to Y$, which is defined as follows:

$$X \to Y := \{u \,|\, \forall v \in X, u@v \in Y\}$$

- If X_i is a nonempty family of realizability predicates, so is $\bigcap_{i\in I} X_i$. In particular, if I is the set of all realizability predicates and $\mathcal{M} : I \to I$ some map between realizablity predicates. Then $\bigcap_{i\in I} \mathcal{M}(i)$ is a realizability predicate.

In the formalization, we see a few minor adjustments that do not surprise us.

```
Definition top_rp u := u ~ top.

Definition prod X Y : realizorCandidate :=
  fun u => IFilter u /\ forall v, X v -> Y (u @ v).

Definition intersect I (X : I -> realizorCandidate) :
  realizorCandidate := fun u => forall i : I, X i u.
Definition possible_int (M : realizorCandidate -> realizorCandidate)
  : realizorCandidate :=
    intersect { X | realizorPredicate X } (fun P => M (proj1_sig P)).
```

And:

```
Lemma top_rp_realizable : realizorPredicate top_rp.

Lemma prod_realizable X Y :
  realizorPredicate X -> realizorPredicate Y ->
    realizorPredicate (prod X Y).

Lemma intersect_realizable I (X : I -> realizorCandidate) :
  inhabited I ->
  (forall i : I, realizorPredicate (X i)) ->
    realizorPredicate (intersect I X).

Lemma possible_int_realizable M :
  (forall X, realizorPredicate X -> realizorPredicate (M X)) ->
    realizorPredicate (possible_int M).
```

Furthermore, we make explicit the fact that we only work on extensional equality. For extensional equality between realizability predicates, we use the symbol `=R=`: [4]

```
Definition rp_equiv X Y := forall u, X u <-> Y u.

Infix "=R=" := rp_equiv (at level 55).
```

[4] The notion of extensional equality should have been generalized.

3.5 Framework.v: Proving Termination of Other Logics

This file defines the main tools which are used to show Termination for other type theories. The first such tool are valuations σ, which are maps from variables to realizability predicates.

```
Definition rc_ctx := var -> realizorCandidate.
Implicit Types sigma : rc_ctx.

Definition valuation sigma := forall x, realizorPredicate (sigma x).
```

Let $\mathbb{T}$ and $\mathbb{D}$ be the set of (syntactically correct) types and terms, respectively, in the type theory we want to consider. In the following, we will use a, b, c for variables ranging over $\mathbb{D}$ and S, T for variables ranging over $\mathbb{T}$. The code matches this perfectly:

```
  Variable ttype : Type.
  Implicit Types S T : ttype.

  Variable tterm : Type.
  Implicit Types a b c : tterm.
```

We consider a quarternary typing relation, $\rho, \sigma \vdash a : T$. This is not standard but very versatile, since it does not make any assumptions about the existence of contexts or variables in the type theory. We define a binary typing relation, where the ρ and σ are existentially quantified:

$$a : T := \exists \rho, \sigma. \rho, \sigma \vdash a : T$$

The formalization currently goes a slightly different way. The quarternary relation (`vtty`) and the binary relation (`ctty`) are both assumed, along with a proof that they behave as mentioned:

```
Variable ctty : tterm -> ttype -> Prop.

Variable vtty : forall rho sigma, tterm -> ttype -> Prop.
Definition validation := forall a T,
  ctty a T <-> (exists rho sigma,
    valid rho /\ valuation sigma /\ vtty rho sigma a T).
```

We also consider a reduction relation $\rightsquigarrow$, of which we want to show that it terminates on all typable terms:

```
Variable tstep : tterm -> tterm -> Prop.
Definition TSN := GSN tstep.
```

The proof goes as follows. We assume a type interpretation $[\![\cdot]\!]_\sigma$ mapping types to realizability predicates, and a compiler $|\cdot|$ mapping terms of the source system to terms of the target system with the following two properties:

Compiler Consistency: If $a \rightsquigarrow b$, then $|a| \rightarrow_\beta^+ |b|$

Adequacy: If $\sigma, \rho \vdash a : T$, then $[|a|]_\rho \in [\![T]\!]_\sigma$

The formalization is straightforward:

```
Variable int_ty : ttype -> rc_ctx -> realizorCandidate.
Notation "[[ T ]] sigma" := (int_ty T sigma) (at level 1).

Definition rp_preservation :=
  (forall T sigma, valuation sigma ->
    realizorPredicate [[T]]sigma).

Variable compiler : tterm -> term.
Notation "[| a |]" := (compiler a) (at level 9).

Definition compiler_consistency := forall a b,
  tstep a b -> steps [|a|] [|b|].

Definition cadequacy := forall {rho sigma},
  valid rho -> valuation sigma ->
    forall {a T}, vtty rho sigma a T -> [[T]]sigma [ [|a|] ]rho.
```

Note that this is enough to show Termination. Due to Adequacy and the fact that realizability predicates contain only non-empty filters, $a : T$ implies $[|a|]_\rho$ is nonempty for some ρ. Therefore $|a|$ has a type in the target system and beta reduction terminates on $|a|$. Due to Compiler Consistency, if beta reduction terminates on $|a|$, then $\rightsquigarrow$ terminates on a.

These facts are formalized as follows:

```
Lemma sn_tsn a : compiler_consistency -> SN [|a|] -> TSN a.

Theorem tsn : validation -> rp_preservation ->
  cadequacy -> compiler_consistency ->
    forall a T, ctty a T -> TSN a.
```

For type theories where $\mathbb{D} = \text{ULC}$[5], the assumptions can be condensed further. As within the target system, we use s, t for variables ranging over ULC. With $|t| = t$, Compiler Consistency comes for free. Furthermore, we make different assumptions on the typing relation. Rather than a quarternary relation that considers ρ and σ, we now treat a standard ternary relation that considers a context Γ that maps variables to types: $\Gamma \vdash_{\mathbb{T}} t : T$.

```
Definition lcc t := t.
Lemma lcc_c : (compiler_consistency _ lcc step).

Definition t_ctx := var -> ttype.
Implicit Types tGamma : t_ctx.
Variable tty : t_ctx -> term -> ttype -> Prop.
```

[5]In the development, this equality is syntactic and thus also includes the representation.

We can now change Adequacy to a weaker form. Since our typing relation itself no longer considers ρ and σ, we need a new way to relate them to our typing judgement. We do so via our context and define:

$$\Gamma \vdash \rho, \sigma := \forall x, [x]_\rho \in [\![\Gamma\ x]\!]_\sigma$$

If $\Gamma \vdash_{\mathbb{T}} x : \Gamma\ x$ holds, this is a subcase of Adequacy as defined before.

In the formalization, we take the liberty of applying the model property for variables ($[x]_\rho = \rho\ x$).

```
Definition validates rho tGamma sigma :=
  forall x, [[tGamma x]]sigma (rho x).
```

This gives rise to the following quarternary typing relation:

$$\rho, \sigma \vdash t : T := \exists \Gamma. \Gamma \vdash t : T \wedge \Gamma \vdash \rho, \sigma$$

The formalization is straightforward:

```
Inductive val_ty rho sigma t T : Prop :=
val_ty_gamma tGamma :
  validates rho tGamma sigma ->
  tty tGamma t T ->
    val_ty rho sigma t T.
```

Adequacy can then be stated as follows: If $\Gamma \vdash \rho, \sigma$ and $\Gamma \vdash_{\mathbb{T}} t : T$, then $[t]_\rho \in [\![T]\!]_\rho$. That this is strong enough to prove the previous Adequacy (w.r.t. the quarternary relation defined above) is clear.

```
Definition adequacy :=
  forall {tGamma t T},
    tty tGamma t T ->
    forall {rho sigma},
      valid rho -> valuation sigma ->
      validates rho tGamma sigma ->
        [[T]]sigma [t]rho.

Lemma adequacy_cadequacy : adequacy -> cadequacy _ lcc val_ty.
```

Consequently, the new formulation of Adequacy implies Termination:

```
Lemma sn :
  rp_preservation ->
  adequacy ->
    forall tGamma t T, tty tGamma t T -> SN t.
```

Chapter 4

Source Systems

4.1 CL1.v: Combinatory Logic

We consider Combinatory Logic without variables, if only to show that the framework is versatile enough to handle such a source system. The term syntax is therefore as follows:

$$a, b ::= \mathrm{S} \mid \mathrm{K} \mid a\ b$$

The type syntax is the same as the simple type system:

$$A, B ::= \alpha \mid A \to B$$

The reduction relation is as follows:

$$\frac{}{\mathrm{S}\ a\ b\ c \rightsquigarrow a\ c\ (b\ c)} \qquad \frac{}{\mathrm{K}\ a\ b \rightsquigarrow a} \qquad \frac{a \rightsquigarrow a'}{a\ b \rightsquigarrow a'\ b} \qquad \frac{b \rightsquigarrow b'}{a\ b \rightsquigarrow a\ b'}$$

The typing relation is binary and straightforward:

$$\frac{}{\mathrm{S} : (A \to B \to C) \to (A \to B) \to A \to C} \qquad \frac{}{\mathrm{K} : A \to B \to A}$$

$$\frac{a : A \to B \qquad b : A}{a\ b : B}$$

These definitions are formalized as follows:

```
Inductive CL : Type :=
| S
| K
| app (a b : CL).
Implicit Types a b c : CL.
Coercion app : CL >-> Funclass.

Inductive simpletype : Type :=
```

```
| base (x : var) : simpletype
| arr (A B : simpletype) : simpletype.
Notation "A --> B" := (arr A B).
Implicit Types A B C : simpletype.

Inductive CL_ty : CL -> simpletype -> Prop :=
| CLS A B C : CL_ty S ((A --> B --> C) --> (A --> B) --> A --> C)
| CLK A B : CL_ty K (A --> B --> A)
| CLapp a b A B : CL_ty a (A --> B) -> CL_ty b A -> CL_ty (a b) B.

Inductive cl_step : CL -> CL -> Prop :=
| cl_stepS a b c : cl_step (S a b c) (a c (b c))
| cl_stepK a b : cl_step (K a b) a
| cl_step_appL a b c : cl_step a b -> cl_step (a c) (b c)
| cl_step_appR a b c : cl_step b c -> cl_step (a b) (a c).
```

The compiler follows the reduction relation[1]:

$$
\begin{aligned}
|\mathrm{S}| &:= \lambda f, g, x. f\ x\ (g\ x) \\
|\mathrm{K}| &:= \lambda x, y. x \\
|a\ b| &:= |a|\ |b|
\end{aligned}
$$

This compiler is obviously consistent.

```
Fixpoint CLC a :=
  match a with
  | S => Lam (Lam (Lam (TVar 2 (TVar 0) (TVar 1 (TVar 0)))))
  | K => Lam (Lam (TVar 1))
  | app a b => CLC a (CLC b)
  end.

Lemma CLC_c : compiler_consistency _ CLC cl_step.
```

We interpret types as follows:

$$
\begin{aligned}
[\![\alpha]\!]_\sigma &:= \sigma\ \alpha \\
[\![A \to B]\!]_\sigma &:= [\![A]\!]_\sigma \to [\![B]\!]_\sigma
\end{aligned}
$$

This is clearly a realizability predicate.

```
Fixpoint int_ty A sigma :=
  match A with
  | base x => sigma x
  | arr A B => prod (int_ty A sigma) (int_ty B sigma)
  end.
```

[1] We use a syntactic shorthand and omit sequential binders

```
Notation "[[ A ]] sigma" := (int_ty A sigma) (at level 1).

Lemma int_ty_rp : rp_preservation simpletype int_ty.
```

We show Adequacy (with $\rho, \sigma \vdash a : A := a : A$) by induction on the typing relation. In order to demonstrate the proof mechanism for Adequacy, we will go through this proof in more detail. This yields three cases:

S : $(A \to B \to C) \to (A \to B) \to A \to C$**:** We have to show:

$$\begin{aligned}[\lambda f, g, x.f\ x\ (g\ x)]_\rho \in (&[\![A]\!]_\sigma \to [\![B]\!]_\sigma \to [\![C]\!]_\sigma) \\ &\to ([\![A]\!]_\sigma \to [\![B]\!]_\sigma) \\ &\to [\![A]\!]_\sigma \to [\![C]\!]_\sigma\end{aligned}$$

This by definition of the functional product is the same as showing that if $u_f \in [\![A]\!]_\sigma \to [\![B]\!]_\sigma \to [\![C]\!]_\sigma$, $u_g \in [\![A]\!]_\sigma \to [\![B]\!]_\sigma$ and $u_x \in [\![A]\!]_\sigma$, then $[\lambda f, g, x.f\ x\ (g\ x)]_\rho @ u_f @ u_g @ u_x \in [\![C]\!]_\sigma$. Since the above are all realizability predicates, $u_f, u_g, u_x \neq \emptyset$ and thus the model properties apply. Therefore:

$$\begin{aligned}[\lambda f, g, x.f\ x\ (g\ x)]_\rho @ u_f @ u_g @ u_x &= [f\ x\ (g\ x)]_{\rho^{f,g,x}_{u_f,u_g,u_x}} \\ &= [f]_{\rho^{f,g,x}_{u_f,u_g,u_x}} @ \cdots @ (\cdots @ \cdots) \\ &= u_f @ u_x @ (u_g @ u_x)\end{aligned}$$

Which is indeed an element of $[\![C]\!]_\sigma$ due to the memberships of u_f, u_g, and u_x mentioned above.

K : $A \to B \to A$**:** This is analogous to the case for S.

$a\ b : B$**:** The inductive hypothesis for $a : A \to B$ is that $[|a|]_\rho @ u \in [\![B]\!]_\sigma$ for all $u \in [\![A]\!]_\sigma$. The claim is $[|a|]_\rho @ [|b|]_\rho \in [\![B]\!]_\sigma$ if $[|b|]_\rho \in [\![A]\!]_\sigma$ and is thus a consequence of the inductive hypothesis.

As can be seen above, the proofs follow by rewriting the model properties and the fact that the filters we apply our abstractions to are always nonempty. The theorem is formalized in `in_int_ty`, and Termination is a corollary.

```
Lemma in_int_ty : cadequacy simpletype int_ty CL CLC (fun _ _ => CL_ty).
Corollary CL_stlc t A : CL_ty t A -> TSN _ cl_step t.
```

4.2 STLC1.v: Simply Typed Lambda Calculus

Simply typed lambda calculus uses ULC. The type syntax is as in CL1.v and types are interpreted in the same way. However, due to the presence of variables, the typing system is ternary:

$$\frac{}{\Gamma \vdash x : \Gamma\ x} \qquad \frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B} \qquad \frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A \to B}$$

```
Inductive simpletype : Type :=
| base (x : var) : simpletype
| arr (A B : simpletype) : simpletype.

Implicit Types A B C : simpletype.

Definition stlc_ctx := var -> simpletype.
Implicit Types Gamma Delta : stlc_ctx.

Inductive stlc_ty : stlc_ctx -> term -> simpletype -> Prop :=
| stlctyvar Gamma x : stlc_ty Gamma x (Gamma x)
| stlctyapp Gamma s t A B : stlc_ty Gamma s (arr A B) ->
  stlc_ty Gamma t A -> stlc_ty Gamma (s t) B
| stlctylam Gamma s A B : stlc_ty (A .: Gamma) s B ->
  stlc_ty Gamma (Lam s) (arr A B).
```

We have to show (the modified form of) Adequacy: if $\Gamma \vdash t : A$ and $\Gamma \vdash \rho, \sigma$, then $[t]_\rho \in [\![T]\!]_\sigma$.

We only show the variable case since the other cases are analogous to the proofs in CL1.v. By assumption, we have $\Gamma \vdash \rho, \sigma$. This is the claim.

The lemma and its corolloary, Termination, are formalized as follows:

```
Lemma in_int_ty : adequacy simpletype int_ty stlc_ty.
Corollary sn_stlc Gamma t A : stlc_ty Gamma t A -> SN t.
```

4.3 T1.v: System $\mathcal{T}$

System $\mathcal{T}$ is STLC extended with an inductive datatype, specifically the natural numbers. Typically Termination proofs of System $\mathcal{T}$ are considered very tedious. Direct proofs with reducibility candidates require one nested induction per argument, and prec (the eliminator of the natural numbers) has three. Proofs that embed System $\mathcal{T}$ into a stronger system like System $\mathcal{F}$ usually embed the numbers via their Church encoding, thus rendering prec very tedious to embed. However, one of the formidable qualities of our target system is that it is capable of typing the Church-Scott encoding of, e.g., the natural numbers. This idea, as we will show, yields an elegant embedding for all three primitives (S, O, and prec) and a short proof of Termination.

The term syntax of System $\mathcal{T}$ is thus:

$$a, b := x \mid \lambda x.a \mid a\ b \mid \mathrm{S} \mid \mathrm{O} \mid \mathrm{prec}$$

The reduction relation is that of ULC plus the following two rules:

$$\frac{}{\mathrm{prec}\ \mathrm{O}\ x\ f \rightsquigarrow x} \qquad \frac{}{\mathrm{prec}\ (\mathrm{S}\ n)\ x\ f \rightsquigarrow f\ n\ (\mathrm{prec}\ n\ x\ f)}$$

The type syntax is like that of STLC, except for the variables which have been replaced by NAT, the type for the natural numbers.

$$A, B ::= \mathrm{NAT} \mid A \to B$$

Finally the typing relation, which adds the following three rules to those of STLC:

$$\frac{}{\Gamma \vdash \mathrm{O} : \mathrm{NAT}} \qquad \frac{}{\Gamma \vdash \mathrm{S} : \mathrm{NAT} \to \mathrm{NAT}}$$

$$\frac{}{\Gamma \vdash \mathrm{prec} : \mathrm{NAT} \to A \to (\mathrm{NAT} \to A \to A) \to A}$$

All of this is captured in the formalization:

```
Inductive Tterm : Type :=
| Tvar (x : var)
| Tlam (a : {bind Tterm})
| Tapp (a b : Tterm)
| Tprec | TO | TS.

Implicit Types a b c : Tterm.
Coercion Tapp : Tterm >-> Funclass.
Coercion Tvar : var >-> Tterm.

Inductive T_step : Tterm -> Tterm -> Prop :=
| Tbeta a b : T_step ((Tlam a) b) (a.[b /])
| TappL a a' b : T_step a a' -> T_step (a b) (a' b)
| TappR a b b' : T_step b b' -> T_step (a b) (a b')
| TprecO b c : T_step (Tprec TO b c) b
| TprecS a b c : T_step (Tprec (TS a) b c) (c a (Tprec a b c)).

Inductive Ttype : Type :=
| NAT
| arr (A B : Ttype).

Implicit Types A B C : Ttype.
Infix "-->" := arr.

Definition T_ctx := var -> Ttype.
Implicit Types Gamma Delta : T_ctx.

Inductive T_ty : T_ctx -> Tterm -> Ttype -> Prop :=
| Ttyvar Gamma x : T_ty Gamma x (Gamma x)
| Ttyapp Gamma a b A B : T_ty Gamma a (A --> B) ->
  T_ty Gamma b A -> T_ty Gamma (a b) B
| Ttylam Gamma a A B : T_ty (A .: Gamma) a B ->
  T_ty Gamma (Tlam a) (A --> B)
| Ttyprec Gamma A :
  T_ty Gamma Tprec (NAT --> A --> (NAT --> A --> A) --> A)
```

```
| TtyO Gamma : T_ty Gamma TO NAT
| TtyS Gamma : T_ty Gamma TS (NAT --> NAT
```

Compiling variables, applications, and abstractions is straightforward. As mentioned before, we compile the numbers as their Church-Scott encoding. Therefore, it suffices to compile prec as the identity function. However, in order for Compiler Consistency to hold, we have to deviate slightly from the Church-Scott encoding and copy the compilate of prec:

$$\begin{aligned} |\mathrm{O}| &:= \lambda x, f.x \\ |\mathrm{S}| &:= \lambda n, x, f.f\ n\ (|\mathrm{prec}|\ n\ x\ f) \\ |\mathrm{prec}| &:= \lambda x.x \end{aligned}$$

In order for this to go formally through, we have to hardcode the compilate of prec:

```
Fixpoint compile a : term :=
  match a with
  | Tvar x => x
  | Tlam a => Lam (compile a)
  | Tapp a b => (compile a) (compile b)
  | Tprec => Lam '0
  | TO => Lam (Lam '1)
  | TS => Lam (Lam (Lam ('0 '2 ((Lam '0) '2 '1 '0))))
  end.
```

In order to show that this compiler is consistent, we first have to show that it is compatible with beta substitution:

$$|a_b^x| = |a|_{|b|}^x$$

This and the fact that the compiler is consistent are formalized in the following two lemmas:

```
Corollary compile_subst a b : compile a.[b/] = (compile a).[compile b/].
Lemma TC_c : compiler_consistency _ compile T_step.
```

For the type interpretation, we need a new instance of realizability predicates for NAT. From the typing relation and Adequacy and the definition of realizability predicates, we get the following requirements:

- $\{\top\} \in [\![\mathrm{NAT}]\!]_\sigma$
- $[|O|]_\rho \in [\![\mathrm{NAT}]\!]_\sigma$
- when $u \in [\![\mathrm{NAT}]\!]_\sigma$, then so is $[|\mathrm{S}|]_\rho @ u$

This suffices as an inductive definition:

```
Inductive int_nat u : Prop :=
| top_nat : u ~ top -> int_nat u
| O_nat : u ~ [compile TO] top_rho -> int_nat u
| S_nat v : int_nat v ->
  u ~ [compile TS]top_rho @ v ->
  int_nat u.
```

However, it remains to show that $u \in [\![\mathrm{NAT}]\!]_\sigma$ implies $u \neq \emptyset$. A direct proof of this property does not go through, since in the case of $[|\mathrm{S}|]_\rho|@u$ we need the parameter to have a certain structure. We consider a type of $|\mathrm{S}|$:

$$(F \to A \to F) \circ (F \to A \to F) \to F \to ((F \to A \to F) \to F \to F) \circ A) \to F$$

Thus, if u contains a type of the form $F \to A \to F$, so does $[|\mathrm{S}|]_\rho@u$, and the following proof goes through:

```
Inductive Ktype u : Prop :=
| inKtype (F : basetype) (A : inttype) : u (F ==> A ==> F) -> Ktype u.

Lemma k_nat u : int_nat u -> Ktype u.
```

Therefore, `int_nat` is a realizability predicate and the following go through:

```
Lemma rp_nat : realizorPredicate int_nat.

Fixpoint int_ty A (sigma : rc_ctx):=
  match A with
  | NAT => int_nat
  | arr A B => prod (int_ty A sigma) (int_ty B sigma)
  end.
Notation "[[ A ]]" := (int_ty A) (at level 1).

Lemma int_ty_rp : rp_preservation Ttype int_ty.
```

For the quarternary typing relation, we use the same as in the generic ULC proof:

```
Inductive val_ty rho sigma t T : Prop :=
val_ty_gamma tGamma :
  validates _ int_ty rho tGamma sigma ->
  T_ty tGamma t T ->
    val_ty rho sigma t T.
```

The proof of Adequacy has one new case, and that is for prec. After rewriting with the model properties, we have to show that if u_n is in the interpretation of NAT, u_x in that of A, and u_f in that of NAT $\to A \to A$, then $u_n@u_x@u_f$ is also in the interpretation of A. The proof of this is by straightforward induction on the proof that u_n is in the interpretation of NAT. After Adequacy is shown Termination follows.

```
Lemma in_int_ty : cadequacy Ttype int_ty _ compile val_ty.
Corollary sn_T Gamma a A : T_ty Gamma a A -> TSN _ T_step a.
```

4.4 F1.v: System $\mathcal{F}$

Due to the polymorphic strength of System $\mathcal{F}$ we were surprised at how straightforward this system is to handle. While the term syntax is that of ULC, the type syntax allows for non-dependent products (i.e., generalization of type variables):

$$S, T ::= \alpha \mid S \to U \mid \Pi\alpha.T$$

This generalization can be instanciated arbitrarily. However generalizing a variable can only be allowed if it does not lead to an inconsistency, i.e., if that variable does not occurs in the context[2] Therefore the following two rules are added to those of STLC:

$$\frac{\alpha \text{ is not free in } \Gamma \qquad \Gamma \vdash t : T}{\Gamma \vdash t : \Pi\alpha.T} \qquad\qquad \frac{\Gamma \vdash t : \Pi\alpha.T}{\Gamma \vdash t : T_S^\alpha}$$

```
Inductive Ftype : Type :=
| tvar (x : var)
| arr (S T : Ftype)
| pi (T : {bind Ftype}).

Definition f_ctx := t_ctx Ftype.
Implicit Types Gamma Delta : f_ctx.

Inductive f_ty : f_ctx -> term -> Ftype -> Prop :=
| fvar Gamma x : f_ty Gamma x (Gamma x)
| fapp Gamma s t S T : f_ty Gamma s (arr T S) ->
  f_ty Gamma t T ->
    f_ty Gamma (s t) S
| flam Gamma s S T : f_ty (S .: Gamma) s T ->
  f_ty Gamma (Lam s) (arr S T)
| fgen Gamma s T :
  f_ty (fun x => (Gamma x).[ren (+1)]) s T
  -> f_ty Gamma s (pi T)
| fspec Gamma s S T :
  f_ty Gamma s (pi T)
  -> f_ty Gamma s (T.[S/]).
```

With the above typing rule in mind, it is clear that for every realizability predicate X, $[\![\Pi\alpha.T]\!]_\sigma \subseteq [\![T]\!]_{\sigma_X^\alpha}$[3]. Thus we define: $[\![\Pi\alpha.T]\!]_\sigma := \bigcap_X [\![T]\!]_{\sigma_X^\alpha}$. Together with the definition we have from STLC, the code for this is as follows:

```
Fixpoint int_ty T sigma :=
  match T with
  | tvar x => sigma x
  | pi T =>  possible_int (fun X=> int_ty T (X .: sigma))
```

[2]For a counter example, consider $x : \alpha \vdash x : \alpha$; a fallacious generalization would then be $x : \alpha \vdash x : \Pi\beta.\beta$.

[3]This is a generalization. We only need $[|t|]_\rho \in [\![\Pi\alpha.T]\!]_\sigma$ implies $[|t|]_\rho \in [\![T_S^\alpha]\!]_\sigma$

```
  | arr T1 T2 => prod (int_ty T1 sigma) (int_ty T2 sigma)
  end.
Notation "[[ T ]] sigma" := (int_ty T sigma) (at level 1).

Lemma int_ty_rp : rp_preservation Ftype int_ty.
```

Due to substitutions occuring in types, we have to prove that interpretations are stable under substitution. A substitution π translates from σ_1 to σ_2 if $[\![\alpha]\!]_{\sigma_1} = [\![\pi\ \alpha]\!]_{\sigma_2}$ for all α. For such substitutions it holds that $[\![T]\!]_{\sigma_1} = [\![\pi\ T]\!]_{\sigma_2}$, as shown in the formalization:

```
Lemma int_ty_ren_equiv T  sigma (r : var -> var) :
  [[ T.[ren r] ]]sigma =R= [[T]] (fun x => sigma (r x)).

Definition translates s sigma1 sigma2 :=
  forall x, [[ tvar x ]] sigma1 =R= [[s x]]sigma2.

Lemma int_ty_subst_equiv T : forall s sigma1 sigma2,
  translates s sigma1 sigma2 -> [[T]] sigma1 =R= [[ T.[s] ]] sigma2.
```

In the Adequacy proof, there are two new cases which we shall handle here.

$\Gamma \vdash t : \Pi\alpha.T$**:** Assume that $\Gamma \vdash \rho, \sigma$, α is not free in Γ, and $\Gamma \vdash t : T$. We have to show that for all realizability predicates X, $[t]_\rho \in [\![T]\!]_{\sigma^\alpha_X}$ Note that $\Gamma \vdash \rho, \sigma^\alpha_X$ and the induction hypothesis for $\Gamma \vdash t : T$ proves the claim.

$\Gamma \vdash t : T^\alpha_S$**:** Assume that $\Gamma \vdash t : \Pi\alpha.T$. Thus by the induction hypothesis, $[t]_\rho \in [\![T]\!]_{\sigma^\alpha_X}$ for all realizability candidates X. We have to show that $[t]_\rho \in [\![T^\alpha_S]\!]_\sigma$. Recall that $[\![T^\alpha_S]\!]_\sigma = [\![T]\!]_{\sigma^\alpha_{[\![S]\!]_\sigma}}$ The claim follows.

This proves Adequacy and thus Termination of System $\mathcal{F}$.

```
Lemma in_int_ty : adequacy Ftype int_ty f_ty.
Corollary sn_F {tGamma t T} : f_ty tGamma t T -> SN t.
```

Chapter 5

Conclusion

5.1 Limitations of the Current Framework

As we hope to have made apparent in the previous chapter, the framework makes Termination proofs for a plethora of logical systems nothing to be afraid of. In fact, up to non-dependent type theories with inductive types, adapting the proof of STLC is straightforward. But can we go further?

Sadly, that seems not to be the case. The target system is strong enough to cover all terminating terms of ULC, and with compilation even of some extensions like System $\mathcal{T}$. However, if we consider a type theory where types are terms, this will include the functional arrow. It could be possible to compile the functional arrow as an application if the type of the arrow is interpreted as $\{\top\}$, but it sounds very wrong and we have not tried.

Furthermore, we do not have an answer for the question on how to interpret a type that includes an application. Many of such type theories use a conversion rule, thus suggesting that type interpretation must be stable under conversion:

$$[\![(\lambda\alpha.T)\ U]\!]_\sigma = [\![T^\alpha_U]\!]_\sigma$$

The most straightforward interpretation would have properties like the following:

$$\begin{aligned} [\![T\ U]\!]_\sigma &= [\![T]\!]_\sigma @ [\![U]\!]_\sigma \\ [\![\lambda\alpha.T]\!]_\sigma @ X &= [\![T]\!]_{\sigma^\alpha_X} \end{aligned}$$

This however requires a definition of $X@Y$. The definition that springs to mind requires that application of types already be a feature of the target system:

$$X@Y := \{v \mid \exists u.u \to v \in X \wedge u \in Y\}$$
$$u \to v := \{A \mid \forall B \in u.A\ B \in v\}$$

This puts any more interesting system out of reach, like F_ω or even more so CC or ECC. It could be - but we did not investigate this - that a slightly modified target system can overcome these hurdles.

We witnessed two other downsides of the framework in Section 4.3. Recall the additional |prec| in $|\mathrm{S}| = \lambda n, x, f.f\ n\ (|\mathrm{prec}|\ n\ x\ f)$. This is due to our statement of Compiler Consistency being so strict: the compilate of the source redex has to reduce to the compilate of the reduct. We believe that there are less strict relations that can replace reduction, but did not find a relation where we could prove that it is still strong enough.

The second downside is how the typing details of the target system are revealed when typing functional constructors in inductive types, like |S|.

Furthermore, the framework is rather large and even though the resulting proofs are very simple and mostly short, the total proof size is still larger with the framework than without it.

5.2 Coquand's Framework

Coquand in [4] uses a parametric programming language for the target system, where constructors as well as bar-recursive operators for inductive datatypes can be added without the need to compile. This has many advantages:

- compilation is not necessary,
- type constructors allow for universes and dependent types,
- no typing properties of the target system are exposed.

Thus, almost all negative aspects of the current framework are negated. That does not mean that the framework comes without its own downsides, two of which we want to discuss here:

The first downside is that the target system is less simple and Termination is no longer a simple consequence of the typing rules. The proof of Termination in Coquand uses a reducibility candidates proof, which can be considered more complex than the proof we formalized.

The second downside strikes us as more severe, since it concerns the use of induction-recursion in the proof of Coquand. Induction-recursion is a formal definition by induction that uses recursion on the type being inductively defined. Sadly, Coq at this point does not support induction-recursion[1]; therefore, this

[1] Some other proof assistants like Agda support inductive-recursive definitions.

proof can not be fully formalized. The line in question is in the definition of T_0[2] and can be stated as follows:

$$\frac{u \in T_0 \qquad \forall x \in El(u).v@x \in T_0}{[\![\mathrm{Fun}]\!]@u@v \in T_0}$$

where El is defined by recursion on T_0[3].

Note that this is not a definition in the framework, but in the definition of the model for a particular source system. Thus if the source system is weak enough, i.e., does not include a dependent product, induction-recursion is simply not needed to define a model:

$$\frac{u \in T_0 \qquad v \in T_0}{u \to v \in T_0}$$

For the stronger system there does not seem to be a way around this definition. Imagine for the sake of argument a definition El' for a larger set $T' \supseteq T_0$, such that for $u \in T_0$, $El'(u) = El(u)$. Then the following rule could be used:

$$\frac{u \in T_0 \qquad \forall x \in El'(u).v@x \in T_0}{[\![\mathrm{Fun}]\!]@u@v \in T_0}$$

However in order to define T', one would need to find a rule for products with two properties:

- whenever $u \in T_0$ and $\forall x \in El'(u).v@x \in T_0$, then $[\![\mathrm{Fun}]\!]@u@v \in T'$ (since otherwise $T' \not\supseteq T_0$), and
- the recursion principle for this rule must allow us to access $v@x$ for all $x \in El'(u)$ (since otherwise $El'([\![\mathrm{Fun}]\!]@u@v) \neq El([\![\mathrm{Fun}]\!]@u@v)$).

The only rule that comes to mind with these properties is the original (inductive-recursive) one.

[2] p. 12 in Coquand [4]

[3] Note that the typing of the target system in Coquand replicates the structure of constructors in the type. Thus the interpretation of Π, U, etc., is injective, and T_0 can be considered an inductive definition.

Bibliography

[1] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. "A filter lambda model and the completeness of type assignment". In: *The journal of symbolic logic* 48.04 (1983), pp. 931–940.

[2] Alexis Bernadet and Stéphane Graham-Lengrand. "Non-idempotent intersection types and strong normalisation". In: *arXiv preprint arXiv:1310.1622* (2013).

[3] Alonzo Church. "A formulation of the simple theory of types". In: *J. Symb. Log.* 5.2 (1940), pp. 56–68.

[4] Thierry Coquand and Arnaud Spiwack. "A proof of strong normalisation using domain theory". In: *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*. IEEE. 2006, pp. 307–316.

[5] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[6] Herman Geuvers. "A short and flexible proof of Strong Normalization for the Calculus of Constructions". In: *TYPES*. 1994, pp. 14–38.

[7] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Vol. 7. Cambridge University Press Cambridge, 1989.

[8] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Clarendon Press, 1994. ISBN: 9780198538356. URL: `http://books.google.de/books?id=z3uicYzJR1MC`.

[9] Gert Smolka, Steven Schäfer, and Tobias Tebbi. "AUTOSUBST: Automation for de Bruijn Substitutions". In: (2014).

[10] William W Tait. "Intensional interpretations of functionals of finite type I". In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212.