Categorical abstract machines for higher-order typed $\lambda$-calculi

Eike Ritter*

Computing Laboratory, Oxford University, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract


Curien's CAM is an environment machine for the untyped $\lambda$-calculus based on cartesian closed categories (CCC's). This categorical model represents both environments and terms by morphisms regardless of their conceptual difference. We show that Ehrhard's D-categories yield a nice way of separating these two notions. Based on suitable categorical combinators for these D-categories we derive an eager and a lazy abstract machine. These machines specialize to the CAM and to Krivine's machine respectively. D-categories extended with additional structure to model the calculus of constructions yield generalizations of the CAM and Krivine's machine to this higher-order $\lambda$-calculus. We also obtain an algorithm for type checking of these combinators, which uses the above reduction machines. Tests using Church-numerals show that the abstract machines are quite efficient compared to other implementations.

1. Introduction

The derivation of categorical abstract machines for a typed $\lambda$-calculus starts with the choice of an appropriate categorical structure. This structure must have an equational presentation, the so-called categorical combinators. These yield a variable-free presentation of the calculus together with an explicit substitution mechanism. Next, one chooses an inference system for the reduction of combinators corresponding to closures. Such a system finally yields an abstract machine in a natural way. This approach leads to simple and conceptually clean combinators that in turn give rise to a simple and conceptually clean instruction set together with an easy correctness proof. The explicit substitution operation makes it possible to postpone substitutions.
during reduction. This considerably improves the efficiency of the machines. Furthermore, the machines have a modular structure, i.e. an extension of the language corresponds to an addition of categorical concepts and machine instructions.

The earliest categorical abstract machine is the CAM, constructed by Cousineau et al. [3]. It handles reduction of closed terms of the untyped $\lambda$-calculus to weak head normal form according to an eager strategy. The relation of the CAM to cartesian closed categories (CCC's) is somewhat problematic because several concepts that are important for the design of abstract machines are not properly modelled in CCC's:

- Environments and terms are both represented by morphisms although they are conceptually different.
- Composition has two roles, namely substitution in a term with respect to environments and application of a function to an argument.
- Product types and contexts are both modelled by products in the CCC, so again two separate issues are merged into one construction.

These mismatches become apparent in the correctness proof of the CAM. The proof in [3] uses various intermediate calculi to describe the explicit substitution, hence it becomes rather complicated.

Jacobs [14] describes a way of turning a CCC into an indexed category that solves the last of these problems. In that approach however, the representation of environments is still unsatisfactory for the design of abstract machines. The reason is that the products in the base category, which model the environments, impose a tree structure on environments although a list structure is sufficient. The $\lambda_2$-calculus [5] adds an explicit notion of environment to the simply typed $\lambda$-calculus to overcome the above three problems. A generalization, the $\lambda\sigma$-calculus [1], uses explicit substitutions to derive first an extension of Krivine's machine describing reduction to normal form for the untyped $\lambda$-calculus and second a type checker for the simply typed and the second-order $\lambda$-calculus. For the case of the simply typed $\lambda$-calculus, the handling of environments in this version of the $\lambda\sigma$-calculus turns out to be quite close to the one described in the approach below. Crégut [4] uses the variant given in [5], which is linked to multicategories, to construct an abstract machine with a different handling of global variables. We propose split D-categories [9], which are particular indexed categories, as an appropriate categorical framework for abstract machines not just for the simply typed $\lambda$-calculus but also for higher-order typed $\lambda$-calculi. In this paper we will concentrate on the calculus of constructions. Split D-categories achieve the separation of terms and environments in a very natural way because terms correspond to morphisms in the fibres and environments to morphisms in the base category. This automatically leads to different combinators for substitution (which is modelled by the reindexing functor) and function application (which corresponds to composition in the fibre). The D-categories model also cartesian products and contexts differently: the former are handled by left adjoints to weakening and the latter by a right adjoint to the terminal object functor.

There are two groups of abstract machines based on these D-categories. The first uses eager environments, i.e. environments that contain only canonical combinators,
whereas in the second an environment may contain arbitrary expressions. Further
design decisions yield machines that, if restricted to the combinators corresponding to
the simply typed \(\lambda\)-calculus, can be transformed directly into the CAM (eager case) or
Krivine's machine (lazy case); see Section 2.4 for details. These abstract machines have
a modular structure, i.e. those for the calculus of constructions contain those for the
typed \(\lambda\)-calculus as submachines. It is also possible to mix eager and lazy constructions,
for example one can construct an eager machine with lazy products. This is another
advantage of separating products from environments.

The original version of the calculus of constructions (CC) adds a special type Prop
of propositions, a type of proofs of a proposition and an impredicative universal
quantification over propositions to the simply typed \(\lambda\)-calculus. This yields dependent
types, so a function space becomes a special case of a dependent product. Coquand
and Huet [2] invented this calculus with the propositions-as-types analogy in mind as
a language for formalizing mathematical proofs where a proposition is valid iff the
type of its proofs is inhabited. Luo [18] adds strong sums to model a program
together with its specification. The presence of dependent types in the calculus of
constructions implies that type checking of CC-expressions may include a reduction
of terms. As an example, let \(p\) be any proposition and take a proof of a proposition
\(\forall x: A. p\), i.e. a term \(t\) of type \(\text{Proof}(\forall x: A. p)\). If it is claimed that the term \(ta\), where \(a\) is of
type \(A\), is a proof of a proposition \(q\), then it must be checked whether the two
types \(p[x/a]\) and \(q\) are convertible. This is done by reducing them to a suitable
normal form. It turns out that such a convertibility test is only necessary when
a dependent base type results or when an abstraction or a projection is type-checked.
This yields a type checking algorithm that can be turned directly into an abstract
machine.

The earliest implementation by Coquand and Huet of the calculus of constructions
represents the syntax of the calculus as a tree with bound variables coded by their de
Bruijn-index and defines a parser, a pretty-printer and a type checker based on this
representation. Huet's constructive engine is an abstract machine for a proof checker,
implemented in CAML [13]. Its characteristic feature is a sharing mechanism be-
tween the substituted term and the original pattern, which uses the exceptions of
CAML. Furthermore, the expansion of definitions is avoided as much as possible
during the equality check. The LEGO theorem prover written by Pollack [22] uses
Luo's version of the calculus of constructions. Harper and Pollack [12] give an
algorithm for this, which is the basis for type checking in LEGO. De Bruijn-indices are
used, and expansion of definitions is delayed as much as possible during type
checking. The implementation of the abstract machines in SML described in this
paper uses SML datatypes to represent the combinators. Hence the abstract machines
can be implemented by pattern matching, together with reference types for sharing.
Tests with Church-numerals shows this machine to be more efficient than the other
two.

The paper is structured as follows. First we explain the general framework and
the construction of abstract machines for the simply typed \(\lambda\)-calculus. Second, we
generalize the machines to the calculus of constructions and describes the type checking algorithm. Afterwards we describe an implementation of our machines in SML and compare their efficiency with LEGO and Coq. Finally we present conclusions and further work.

2. The general framework

The simply typed $\lambda$-calculus over a set $\mathcal{G}$ of ground types suffices to explain the general structure of the reduction machines. The exposition follows the way such machines are constructed, so we start with the categorical structure and the combinators, then explain the reduction strategies and finally describe the corresponding abstract machines.

2.1. The combinators

As discussed in the introduction, a categorical semantics that does not identify environments and terms is the key to a better categorical interpretation of abstract machines. We use Ehrhard's notion of a full constant split D-category [9]. Such a structure is a special kind of indexed category $E: \mathcal{B}^{op} \rightarrow \text{Cat}$ such that every category $E(\Gamma)$ has the same set $T$ of objects including a terminal object $1$ and $f^* A = A$ for every morphism $f: \Gamma \rightarrow A$ in $\mathcal{B}$ and $A \in T$, where $f^*$ is the usual abbreviation for the functor $E(f)(\cdot)$. An object $\Gamma$ in the base category $\mathcal{B}$ corresponds to a context, an object $A$ in any fibre $E(\Gamma)$ to a type, a morphism from $1$ to $A$ in $E(\Gamma)$ to a term $t$ of type $A$ in context $\Gamma$, and a morphism from $\Gamma$ to $A$, called a context morphism, to a list of terms in context $\Gamma$. Such a morphism models what functional programmers call an environment, and because the functor $\ast$ captures substitution of a term with respect to an environment, a morphism $f^* t$ corresponds to a closure.

The Grothendieck construction is the key to a compact formulation of a D-category below. It takes an indexed category $E: \mathcal{B}^{op} \rightarrow \text{Cat}$ and produces a fibration which denoted by $p: \text{Gr}(E) \rightarrow \mathcal{B}$. The objects of $\text{Gr}(E)$ are pairs $(\Gamma, A)$, where $\Gamma$ is an object of $\mathcal{B}$ and $A$ is an object of $E(\Gamma)$. A morphism from $(\Gamma, A)$ to $(A, B)$ is a pair $(f, t)$ of morphisms with $f$ a morphism from $\Gamma$ to $A$ in $\mathcal{B}$ and $t$ a morphism from $A$ to $E(f)(B)$ in $E(\Gamma)$. The functor $p$ is the projection to the first component, which maps an object $(\Gamma, A)$ in $\text{Gr}(E)$ to $\Gamma$ and every morphism $(f, t)$ to $f$.

The precise definition is as follows:

**Definition 1.** Let $\mathcal{B}$ be a category with a terminal object $[\cdot]$ and $E: \mathcal{B}^{op} \rightarrow \text{Cat}$ an indexed category over $\mathcal{B}$, i.e. a contravariant functor from $\mathcal{B}$ to $\text{Cat}$, the category of small categories and functors. $E$ is a full constant split D-category over the set of ground types $\mathcal{G}$ if it satisfies
There exists a set \( \mathcal{F} \cong \mathcal{G} \), which is the set of objects of every fibre \( E(\Gamma) \). For every element \( A \) of \( \mathcal{F} \) and every morphism \( f \) in the base category \( \mathcal{B} \), we have \( E(f)(A) = A \). We write \( f^* \) for \( E(f) \) in the sequel.

The set \( \mathcal{F} \) contains an element \( 1 \) that is a terminal object for each fibre \( E(\Gamma) \).

There exists a right adjoint \( G \) to the functor \( I : \mathcal{B} \to Gr(E) \), where \( I \) is defined by

\[
I(\Gamma) = (\Gamma, 1) \quad (\Gamma \in \text{Obj}(\mathcal{B}))
\]

\[
I(f) = (f, \text{Id}) \quad (f \in \text{Hom}(\Gamma', \Gamma)),
\]

We abbreviate \( G((\Gamma, A)) \) to \( \Gamma \cdot A \) and \( G((f, t)) \) to \( f \cdot t \) later on. Furthermore \( (\text{Fst}, \text{Snd}) : (\Gamma \cdot A, 1) \to (\Gamma, A) \) denotes the counit of this adjunction.

For every pair of morphisms \( g : \Gamma \cdot A \to \Delta \cdot B \) and \( f : \Gamma \to \Delta \) such that the diagram

\[
\begin{array}{ccc}
\Gamma \cdot A & \xrightarrow{g} & \Delta \cdot B \\
\text{Fst} \downarrow & & \downarrow \text{Fst} \\
\Gamma & \xrightarrow{f} & \Delta
\end{array}
\]

commutes, there exists a unique morphism \( t : 1 \to B \) in \( E(\Gamma \cdot A) \) such that \( g = f \cdot t \).

For every object \( \Gamma \) of \( \mathcal{B} \) and \( A \) of \( E(\Gamma) \), the functor

\( \text{Fst}^*_* : E(\Gamma) \to E(\Gamma \cdot A) \)

has a right adjoint

\( \Pi_A : E(\Gamma \cdot A) \to E(\Gamma) \).

We will write in the sequel \( \text{Cur} \) for the natural isomorphism between \( \text{Hom}_{E(\Gamma \cdot A)}(B, C) \) and \( \text{Hom}_{E(\Gamma)}(B, \Pi_A(C)) \).

The Beck–Chevalley-condition for the adjunctions \( \text{Fst}^*_* \vdash \Pi_A \) is satisfied in the strict sense, i.e. the equations

\[
f^*(\text{Cur}_A(t)) = \text{Cur}_A((f \cdot \text{Id})^*(t))
\]

hold for every \( f : \Delta \to \Gamma \), \( A \in E(\Gamma) \), \( B \in E(\Gamma \cdot A) \).

Remarks

- The right adjoint \( G \) to \( I \) models substitution and context extension as follows. First, the object \( \Gamma \cdot A \) corresponds to the context \( (\Gamma, x : A) \). Second, if \( t \) is a term of type \( A \) in context \( \Gamma \) and \( u \) is a term of type \( B \) in context \( (\Gamma, x : A) \), then the morphism \( \langle \text{Id}, t \rangle^* u : 1 \to B \) in \( E(\Gamma) \) corresponds to the term \( u[x \backslash t] \). Third, the weakening operation is modelled by the functor \( \text{Fst}^* : E(\Gamma) \to E(\Gamma \cdot A) \).

- The intuition behind the condition (iv) becomes clear if we reformulate it. Because every morphism \( (f, t) : (\Gamma, A) \to (\Delta, B) \) in \( Gr(E) \) is equal to \( (\text{Id}, t) ; (f, \text{Id}) \), it is enough to require fullness only in the special case \( f = \text{Id} \). In this case it means that any morphism \( t : A \to B \) in \( E(\Gamma) \) corresponds uniquely to the morphism \( t' = \text{Snd} ; \text{Fst}^* t : 1 \to B \) in \( E(\Gamma \cdot A) \). It therefore shows how to translate any morphism \( t : A \to B \) in \( E(\Gamma) \) in a unique way to a term of type \( B \) in context \( (\Gamma, A) \).

\[
\]

\[
\]

\[
\]
Condition (v) is the standard way of modelling function types categorically, namely as right adjoints to weakening (see [17] for the simply typed $\lambda$-calculus and [25] for the polymorphic $\lambda$-calculus).

Because this definition uses only indexed categories and adjunctions, standard means (cf. [19, Theorem IV.2]) produce an equational presentation. However, the combinators obtained in this way are not adequate for the construction of abstract machines. First, the $\eta$-rule in the $\lambda$-calculus and the surjective pairing are normally not part of the calculus to be implemented, and many applications of the calculus of constructions for theorem proving and theory abstraction do not need the $\eta$-rule or the subjective dependent sums [22, 18] either. The combinatorial counterparts of these rules cannot be reformulated in such a way that an inspection of the structure of a combinator suffices to check their applicability. This makes the reduction machine a lot more complicated. Therefore we restrict ourselves to a $\lambda$-calculus and to a version of the calculus of constructions without $\eta$-reductions or surjective dependent sums. As a consequence we omit the corresponding equations in the equational theory of categorical combinators as well. It is however possible to extend the machine to compute long $\beta\eta$-normal forms, as needed for example in [21].

Second, the equation $t = !$, where $!$ is the combinator for the unique morphism from $A$ to $1$, is only true if $t$ is a morphism from $A$ to $1$. When we later turn the equations into reduction rules, this means that the reduction $t \rightarrow !$ is subject to a typing constraint. The abstract machines, which are based on these reduction rules, would then have to maintain type information during the reduction, which causes substantial overhead. This can be avoided if the isomorphisms $\Gamma \cdot 1 \cong \Gamma$, $\Pi(1, B) \cong B$, and $\Pi(A, 1) \cong 1$ are treated as identities because in this case the only combinators with domain $A$ and codomain $1$ are the combinators $t; !$ if $A \neq 1$ and $!d$ and $t; !$ if $A = 1$. Hence we need additional sorts for the terminal object and therefore extra kinds of judgements for the morphisms having it as domain or codomain. Although we can construct machines that incorporate the reduction rules for combinators $t : A \rightarrow 1$, this separation suggests an even easier solution: omit them, especially because the simply-typed $\lambda$-calculus has no terminal object anyway. Third, the combinator $\_!$, which captures the fullness condition (iv) above, is not needed for the translation of $\lambda$-calculus-expressions. We therefore omit it as well and obtain the following signature, where $G$ denotes any element of $\mathcal{G}$:

$$\Gamma ::= [ ] \mid \Gamma \cdot A$$ $$f ::= \langle \rangle \mid !d \mid f; f \mid Fst \mid \langle f, t \rangle$$ $$A ::= G \mid \Pi(A, A)$$ $$t ::= !d \mid t; t \mid f; t \mid Snd \mid \text{Cur}(A, t) \mid \text{App}$$
The judgements are

\[
\Gamma \in \text{Obj} \quad \text{\(\Gamma\) is a context}
\]

\[
\Gamma \triangleright \! f : \Gamma' \quad f \text{ is a context morphism from } \Gamma \text{ to } \Gamma'
\]

\[
\Gamma \triangleright A \quad A \text{ is a type in context } \Gamma
\]

\[
\Gamma \triangleright t : A \rightarrow A' \quad t \text{ is a morphism from } A \text{ to } A' \text{ in context } \Gamma
\]

\[
\Gamma \triangleright t : 1 \rightarrow A \quad t \text{ is a morphism from } 1 \text{ to } A \text{ in context } \Gamma
\]

The rules for well-formed combinators are given in Table 1. Because the construction of abstract machines requires reduction rules, we give their definition (cf. Table 2) instead of presenting the equations, which are the reflexive and transitive closure of the reductions. This and all the following tables will later be adapted to the calculus of constructions. They contain therefore the more general case and indicate the specializations to the more restricted calculus.

Ehrhard defines combinators for D-categories as well [10]. They are an intermediate step between the calculus in de Bruijn-form and the categorical combinators presented here. He takes that calculus and replaces the Bruijn-numbers by combinators for explicit substitution. These are derived with the split D-categories in mind, although he only presents the equations and does not discuss their relation to the categorical structure. This implies that he has only the judgement \(\Gamma \triangleright t : A\) and not \(\Gamma \triangleright t : A \rightarrow B\), as in our approach. He does not define a notion of reduction for these combinators, nor does he give an abstract machine for them. The combinators do not admit the way of deriving categorical abstract machines presented in this paper because of the different representation of environments. Ehrhard has no basic combinator that corresponds to an environment \(\langle f, t \rangle\) but uses instead of combinator \((t \mapsto t) \circ f\), where \((t \mapsto t)\) corresponds to \(\langle \text{Id}, t \rangle\) and \(f\) to \(\langle \text{Fst}; f; \text{Snd} \rangle\) in this setting. His combinators make it impossible to represent environments as lists (with additional sharing in the lazy case). This representation is adopted in this paper.

Now we turn to the relation between the combinators, the D-categories and the \(\lambda\)-calculus. In categorical logic, the correspondence between a categorical structure and a type theory is usually shown by constructing an initial category out of the syntax of the latter. The combinators make it possible to reorganize this proof. First, the combinators are shown to be equivalent to the type theory, and then second the addition of the terminal object and the fullness combinator makes it possible to construct an initial category from the combinators. Because the combinators are an equational presentation of the categorical structure, the last step is obvious. So the above correspondence is an easy corollary of the equivalence between the combinators and the calculus. The latter is given by two translations \([\_]\) and \((\_)^c\), one from the \(\lambda\)-calculus into the combinators and the other in the reverse direction. The translation in the first direction is defined as follows:
Table 1
Well-formed combinator expressions

To get the combinators for the typed \(\lambda\)-calculus, replace all combinators \(\langle f, t[A] \rangle\) and \(f \cdot A\) by \(f \cdot t\) and \(A\) respectively and delete rules marked with \((\ast)\). The rule for \(G e W\) applies for the \(\lambda\)-calculus only.

**Contexts**

\[
\begin{array}{c c c}
\Gamma \vdash A & & \Gamma \cdot A \in \text{Obj} \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \in \text{Obj} & & \langle \cdot \rangle : \Gamma \rightarrow \emptyset \\
\end{array}
\]

**Context morphisms**

\[
\begin{array}{c c c}
\Gamma \in \text{Obj} & & \Gamma \cdot A \in \text{Obj} \\
\end{array}
\]

\[
\begin{array}{c c c}
\langle \rangle : \Gamma \rightarrow \emptyset & & \text{ld} : \Gamma \rightarrow \Gamma \\
\end{array}
\]

\[
\begin{array}{c c c}
f : \Gamma \rightarrow \Gamma' & & g : \Gamma' \rightarrow \Gamma'' \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash A & & \Gamma \cdot A \in \text{Obj} \\
\end{array}
\]

\[
\begin{array}{c c c}
f \cdot g : \Gamma \rightarrow \Gamma'' & & \text{Fst} : \Gamma \cdot A \rightarrow \Gamma \\
\end{array}
\]

\[
\begin{array}{c c c}
f \cdot \Gamma \rightarrow \Gamma' & & \Gamma \vdash A' \\
\end{array}
\]

\[
\begin{array}{c c c}
f \cdot \Gamma \rightarrow \Gamma' & & \Gamma \vdash 1 : \Gamma \rightarrow \Gamma \\
\end{array}
\]

**Types**

\[
\begin{array}{c c c}
f : \Gamma \rightarrow \Gamma' & & \Gamma \vdash A' \cdot (\ast) \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \cdot A \cdot B & & \Gamma \vdash \Pi(A, B) \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \in \text{Obj} & & \Gamma \vdash G \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash \Omega \cdot (\ast) & & \emptyset \cdot \Gamma \vdash \Gamma (\ast) \\
\end{array}
\]

**Morphisms**

\[
\begin{array}{c c c}
\Gamma \vdash A & & \Gamma \vdash A : A \rightarrow B \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash s : B \rightarrow C \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash t : A \rightarrow B & & \Gamma \vdash t : A \rightarrow C \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash f \cdot s : A \rightarrow (\ast) \\
\end{array}
\]

\[
\begin{array}{c c c}
f \cdot \Gamma \rightarrow \Gamma' & & \Gamma \vdash A \cdot (\ast) \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash t : A \rightarrow A' & & \Gamma \vdash A : (\ast) \\
\end{array}
\]

\[
\begin{array}{c c c}
f \cdot \Gamma \rightarrow \Gamma' & & \Gamma \vdash f \cdot A \rightarrow f \cdot A' \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash A & & \Gamma \vdash A : (\ast) \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash \text{Snd} : 1 \rightarrow \text{Fst} \cdot A \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash A \cdot \text{App} : \text{Fst} \cdot A \rightarrow \Pi(A, B) \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash A \cdot \text{App} : \Pi(A, B) ightarrow B \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash A \cdot A \rightarrow B & & \Gamma \vdash B \cdot B \rightarrow B \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \vdash B \cdot B \rightarrow B \\
\end{array}
\]

**Definition 2** *(Translation into combinators).* The translation of raw types, terms and contexts into categorical combinators is given by the following function \([\cdot]\), where \(\text{Fst}^n\) is an abbreviation for \(\underbrace{\text{Fst} \cdot \text{Fst} \cdot \cdots \cdot \text{Fst}}_{n\text{-times}} : \)

\[
\begin{array}{c c c}
\Gamma \cdot (\Gamma, A) & & \Gamma \cdot [A] \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \cdot [1] & & \Gamma \cdot [\emptyset] \\
\end{array}
\]

\[
\begin{array}{c c c}
\Gamma \cdot A \cdot B & & \Gamma \cdot B \cdot (\ast) \\
\end{array}
\]
The reduction relation $\rightarrow$ is the smallest relation that is compatible with the combinators (i.e., it satisfies for example $A \rightarrow A'$ implies $\text{Cur}(A, t) \rightarrow \text{Cur}(A', t)$). There is one exception: the rule $A \rightarrow A'$ implies $f \cdot A \rightarrow f \cdot A'$ applies only if $A \neq T$. The rules for the $\lambda$-calculus are the rules of the first four items with $\langle f, t[A] \rangle$ replaced by $\langle f, t \rangle$ and all reduction rules for types replaced by $f \cdot A \rightarrow A$.

(1) Indexed category

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f; \langle \rangle \rightarrow \langle \rangle$</td>
<td>$f; \text{Id} \rightarrow f$</td>
</tr>
<tr>
<td>$\text{Id}; f \rightarrow f$</td>
<td>$f; (g; h) \rightarrow (f; g); h$</td>
</tr>
<tr>
<td>$\text{Id} \cdot A \rightarrow A$</td>
<td>$\text{Id} \cdot t \rightarrow t$</td>
</tr>
<tr>
<td>$f \cdot \text{Id} \rightarrow \text{Id}$</td>
<td>$f \cdot (t; s) \rightarrow (f \cdot t); (f \cdot s)$</td>
</tr>
<tr>
<td>$f \cdot (g \cdot A) \rightarrow (f; g) \cdot A$</td>
<td>$f \cdot (g \cdot t) \rightarrow (f; g) \cdot t$</td>
</tr>
<tr>
<td>$t; \text{Id} \rightarrow \text{Id}$</td>
<td>$t; t \rightarrow t$</td>
</tr>
<tr>
<td>$u; (t; s) \rightarrow (u; t); s$</td>
<td></td>
</tr>
</tbody>
</table>

(2) Adjunction $I + G$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle f, t[A'] \rangle; \text{Fst} \rightarrow f$</td>
<td>$\langle f, t[A'] \rangle \cdot \text{Snd} \rightarrow t$</td>
</tr>
<tr>
<td>$f; \langle g, t[A'] \rangle \rightarrow \langle f; g, f \cdot t[A'] \rangle$</td>
<td></td>
</tr>
</tbody>
</table>

(3) Dependent products

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f \cdot \text{Cur}(A, t); g \cdot \text{App} \rightarrow \langle f, g \cdot \text{Snd}[A] \rangle \cdot t$</td>
<td></td>
</tr>
<tr>
<td>$f \cdot \text{Cur}(A, t) \rightarrow \text{Cur}(f \cdot A, \langle \text{Fst}; f, \text{Snd}[A] \rangle \cdot t)$</td>
<td></td>
</tr>
<tr>
<td>$f \cdot \Pi(A, B) \rightarrow \Pi(f \cdot A, \langle \text{Fst}; f, \text{Snd}[A] \rangle \cdot B)$</td>
<td></td>
</tr>
<tr>
<td>$f \cdot \text{App} \rightarrow \langle \text{Id}, f \cdot \text{Snd} \rangle \cdot \text{App}$</td>
<td></td>
</tr>
</tbody>
</table>

(4) Universal quantification

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h \cdot \forall (A, t) \rightarrow \forall (h \cdot A, (h \cdot \text{Id}[A]) \cdot t)$</td>
<td></td>
</tr>
<tr>
<td>$\langle \langle \rangle, \forall (A, t)[\Omega] \rangle \cdot T \rightarrow \Pi(A, \langle \langle \rangle, t[\Omega] \rangle \cdot T)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Fst}^t \cdot T \rightarrow \langle \langle \rangle, \text{Snd}[\Omega] \rangle \cdot T$</td>
<td></td>
</tr>
<tr>
<td>$\text{Id} \cdot T \rightarrow \langle \langle \rangle, \text{Snd}[\Omega] \rangle \cdot T$</td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow \langle \langle \rangle, \text{Snd}[\Omega] \rangle \cdot T$</td>
<td></td>
</tr>
</tbody>
</table>

Types

$[G] = G \quad [\Pi A.B] = \Pi([A], [B])$

Terms

$[x_n] = \text{Fst}^* \cdot \text{Snd} \quad [\lambda A.t] = \text{Cur}([A], [t]) \quad [ts] = [t]; \langle \text{Id}, [s] \rangle \cdot \text{App}$

The clause for a variable $x_n$ requires that we consider a variable $x_n$ in a context $(x_n: A_m \ldots, x_0: A_0)$. An alternative formulation with de Bruijn-numbers [23] does not
have this context dependency, but is more difficult to read. Therefore it is not used here.

The definition of a translation in the other direction is more complex. There is no translation that respects the judgements and is the inverse of the translation of \( \lambda \)-expressions into combinators. The reason is that for any \( \lambda \)-term \( t \) the combinator \([t]\) is a morphism with domain 1. But if we restrict ourselves to those morphisms, then such an inverse translation can be given. The fullness condition ensures that it can be uniquely extended to a translation of all combinators such that if \( s \) is the translation of any combinator \( t \) with \( \Gamma \triangleright t : A \rightarrow B \) and \( A \neq 1 \), the combinator \([s]^{f} \) is equal to \( t \). The translation of the combinators \( \text{Fst} \) and \( \text{Id} \) poses another problem. The length of the list of \( \lambda \)-terms corresponding to these combinators depends on the length of the context in which their well-formedness is derived. This implies that the translation cannot be defined on raw expressions alone but it needs a parameter indicating the length of that context.

**Definition 3 (Inverse translation).** The translation function \( \langle \cdot \rangle^{f} \) is defined by induction over the structure of those raw combinator that may appear in a morphism \( t : 1 \rightarrow B \) as follows, where \( n \) denotes a natural number:

1. **on contexts**
   
   \[
   \langle \cdot \rangle^{(n, \cdot)} = \langle \cdot \rangle
   \]

2. **on context morphisms**
   
   \[
   \langle \langle \cdot \rangle \rangle^{(n, \langle \cdot \rangle)} = \langle \cdot \rangle
   \]
   
   \[
   \langle \langle \text{Id} \rangle \rangle^{(n, \text{Id})} = \{ n-1, \ldots, 0 \}
   \]
   
   \[
   \langle \langle \text{Fst} \rangle \rangle^{(n, \text{Fst})} = \{ n-1, \ldots, 1 \}
   \]
   
   \[
   \langle \langle \cdot \rangle \rangle^{(n, \cdot)} = \{ t_{m-1}, \ldots, t_{0} \}
   \]
   
   \[
   \langle \langle \text{Fst} \rangle \rangle^{(n, \cdot)} = \langle \langle \text{Fst} \rangle \rangle^{(n, \cdot)}
   \]
   
   \[
   \langle \langle \cdot \rangle \rangle^{(n, \cdot)} = \{ (n, f)^{f}, (n, t)^{f} \}
   \]

3. **on types**
   
   \[
   \langle \langle \cdot \rangle \rangle^{(n, \cdot)} = \{ t_{m-1}, \ldots, t_{0} \}
   \]
   
   \[
   \langle \langle \cdot \rangle \rangle^{(n, f \ast A)} = \langle \langle \cdot \rangle \rangle^{(n, f \ast A)}
   \]
   
   \[
   \langle \langle \cdot \rangle \rangle^{(n, \Pi (A, B))} = \Pi (n, A)^{f} \cdot (n+1, B)^{f}
   \]
(4) on morphisms

\[(\text{Id}) \quad (n, f \cdot \text{Id}; s)^f = (n, s)^f\]
\[ (n, \text{Id}; s)^f = (n, s)^f\]
\[ (n, t; f \cdot \text{Id})^f = (n, t)^f\]
\[ (n, t; \text{Id})^f = (n, t)^f\]

\[ (:) \quad (n, t; (s; u))^f = (n, (t; s); u)^f\]
\[ (n, t; f \cdot (s; u))^f = (n, (t; f \cdot s); f \cdot u)^f\]

\[ (*) \quad \frac{(n, f)^f = \{ t_{m-1}, \ldots, t_0 \}}{(n, f \cdot t)^f = (m, t)^f[x_i \setminus t_i]}\]

(Snd) \quad (n, \text{Snd})^f = 0

(Cur) \quad (n, \text{Cur}(A, t))^f = \lambda(n, A)^f \cdot (n + 1, t)^f

(App) \quad (n, t; f \cdot \text{App})^f = (n, t)^f(n, f \cdot \text{Snd})^f

\[ (r, t; \text{App})^f = (n, t, \text{Id} \cdot \text{App})^f\]

We shall prove in Section 3 that both translations respect the judgements and are inverse to each other. Using these results, we can extend the translation \((-)^f\) to a translation from all combinators to \(\lambda\)-expressions by defining \((n, t)^f\) to be \((n + 1, \text{Snd}; \text{Fst} \cdot t)^f\) for any combinator \(t\) such that \(\Gamma \triangleright t : A \rightarrow B\) with \(A \neq 1\). Theorem 13 implies with \(n = |\Gamma|\) that

\[ (n + 1, \Gamma \cdot A)^f \vdash (n + 1, \text{Snd}; \text{Fst} \cdot t)^f : (n, B)^f\]

and Theorem 14 yields

\[ \Gamma \triangleright t = \left[ (n, t)^f \right]^f : A \rightarrow B\]

Furthermore, any extension of \((-)^f\) that satisfies the last equation has to be defined in this way:

\[ \Gamma \triangleright t = \left[ (n, t)^f \right]^f \]
\[ \Rightarrow \quad \Gamma \cdot A \triangleright \text{Snd}; \text{Fst} \cdot t = \text{Snd}; \text{Fst} \cdot \left[ (n, t)^f \right]^f = \left[ (n, t)^f \right] \]
\[ \Rightarrow \quad (n + 1, \Gamma \cdot A)^f \vdash (n + 1, \text{Snd}; \text{Fst} \cdot t)^f = n + 1, \left[ (n, t)^f \right]^f = (n, t)^f\]

2.2. Reduction rules

We examine now to the reduction rules in detail with the intention of deriving strategies for performing the reductions later on. We will afterwards show how these strategies lead directly to abstract machines.
We aim for a reduction relation that is strongly normalizing and whose normal forms are the translations of $\lambda$-expressions in normal form. The obvious first approach is to orient the equations obtained from the categorical structure. In most cases there is only one sensible way of doing this. For example if we orient the equation $\langle f, t \rangle \ast \text{Snd} = t$ from right to left, the reduction will certainly not be strongly normalizing. The exceptions are the $\eta$-like equations as $\langle \text{Fst}, \text{Snd} \rangle = \text{Id}$ and $\langle \rangle = \text{Id} : [\ ] \rightarrow [\ ]$. If they are treated as rules for simplification of expressions, we have to add some more rules to achieve confluence [7]. One such rule is $\langle f ; \text{Fst}, f \ast \text{Snd} \rangle \rightarrow f$, which requires a test for syntactic equality to check its applicability. This makes the reduction machines extremely inefficient. We will follow therefore more or less the approach of Jay [15] for the equality check of combinators. He thinks of these rules as expansion rules, applied after reduction to $\beta$-normal form.

Remark. In general, we will use the following terminology for reduction according to [16]:

- The one-step reduction is denoted by $t \rightarrow t'$, called “$t$ reduces to $t$”.
- The equivalence relation $\leftrightarrow^*$ generated by $\rightarrow$ is called convertibility, and we say “$t$ is convertible to $t'$” for $t \leftrightarrow^* t'$.
- $t \equiv t'$ denotes the syntactical identity of raw combinators.

The explicit substitution causes a serious problem when we try to prove strong normalization of the reduction $\rightarrow$. The reason are the rules like $f \ast \text{Cur}(A, t) \rightarrow \text{Cur}(f \ast A, \langle \text{Fst}; f, \text{Snd} \rangle \ast t)$, which intuitively correspond in the calculus to pushing substitution inside binding operations like $\lambda$. They cause the usual reducibility approach to fail because in all its variants the reducibility of $\text{Cur}(A, t)$ requires a lemma similar to the following conjecture

**Conjecture 4.** If $s$ is strongly normalizing and $g \ast t$ is strongly normalizing for any strongly normalizing $g$, then also

$$u := f \ast \text{Cur}(A, t) : \langle \text{Id}, s \rangle \ast \text{App}$$

is strongly normalizing.

To see where the problem is, consider the following reduction sequence:

$$u \rightarrow \text{Cur}(A, \langle \text{Fst}; f, \text{Snd} \rangle \ast t) ; \langle \text{Id}, s \rangle \ast \text{App})$$

$$\rightarrow \langle \text{Id}, s \rangle ; \langle \text{Fst}; f, \text{Snd} \rangle \ast t$$

There seems to be no way of deducing from the hypotheses of the lemma that the context morphism $\langle \text{Id}, s \rangle ; \langle \text{Fst}; f, \text{Snd} \rangle$ is strongly normalizing, so the proof of the conjecture breaks down. On the other hand I have not found any nonterminating reduction sequence. It is only known that the reduction corresponding to substitution alone is strongly normalizing [6, 11].
The solution proposed in [23] and reported in [24] for the case of the $\lambda\sigma$-calculus is to restrict the reduction in such a way that substitution is pushed only under the outermost binding operation and not under arbitrary ones. Then Conjecture 4 becomes true because the problematic reduction sequence is no longer permitted; we have only

$$u \rightarrow (f, s) * t$$

and the latter is strongly normalizing by assumption. This intuition is captured by two reductions $\rightarrow^W$ and $\rightarrow^N$. The first describes the reduction to a combinator without pushing substitution inside binding operations, and the latter the reduction of those combinators to normal form. Hence the reduction $\rightarrow^W$ is defined as $\rightarrow^N$ minus the reduction rule

$$f * \text{Cur}(A, t) \rightarrow \text{Cur}(A, \langle\text{Fst}; f, \text{Snd}\rangle * t)$$

The reduction relation $\rightarrow^N$ is the smallest relation including $\rightarrow^W$ plus the above rule and satisfying all of the congruence conditions of $\rightarrow^N$ except the congruence conditions for the combinators $*$ and $;$. For these combinators, we have only the congruence rules

$$f \rightarrow^N f' \quad f; \text{Fst} \rightarrow^N f'; \text{Fst} \quad f \rightarrow^W f' \quad f \rightarrow^N f'; \text{Snd}$$

$$u \rightarrow^N u'$$

$$f * \text{App} \rightarrow^N \langle\text{Id}, f * \text{Snd}\rangle * \text{App} \quad \text{Fst}^k * \text{Snd} ; t ; u ; s \rightarrow^N \text{Fst}^k * \text{Snd} ; t ; u' ; s$$

2.3. Reduction strategies

Abstract machines require not only a notion of reduction but also a strategy for choosing which reduction to execute next. The intuition behind the strategies given below is to describe how a combinator of the form $\langle\langle, t_1, \ldots, t_0\rangle * t$ that corresponds to the substitution of the environment $\{(t_1)^c, \ldots, (t_0)^c\}$ in the term $(t)^c$ can be reduced to a normal form. More precisely, a combinator is reduced according to them first to a so-called canonical one, which is a combinator corresponding to a substitution of an environment in a translation of a $\lambda$-expression in weak head-normal form. The second step, namely the reduction of a canonical combinator to its normal form, proceeds by pushing the substitution inside the binding operation and applying the first step again. Because $\rightarrow^N$ is strongly normalizing, these procedures always terminate with a normal form. Therefore it is enough to describe strategies for reduction to canonical combinators.

Three factors play a key role in the selection of a reduction strategy:

- Evaluating the combinators $f$ and $t$ inside $\langle f, t \rangle$ or not. The first choice corresponds to an eager strategy, where every canonical context morphism $\langle\langle, t_1, \ldots, t_0\rangle$ always contains canonical morphisms $t_i$, and the second to a lazy one with possibly unevaluated expressions $t_i$. 

Table 3
Canonical combinators for the lazy reduction

The rules for types apply only in the case of the calculus of constructions. Furthermore, for canonical combinators for constant D-categories replace \( \langle f, t[A] \rangle \) by \( \langle f, t \rangle \) and \( f \cdot A \) by \( A \).

**Context morphisms**

\[
\begin{array}{c}
\langle \rangle \in \mathcal{C} \\
\text{Fst}^k; \langle f, t[A] \rangle \in \mathcal{C} \\
\text{Id} \in \mathcal{C} \\
f; \langle g, t[A] \rangle \in \mathcal{C} \\
f; \langle g, t[A] \rangle; \langle h, t[B] \rangle \in \mathcal{C}
\end{array}
\]

**Types**

\[
\begin{array}{c}
f \in \mathcal{C} \\
f \cdot \Pi(A, B) \in \mathcal{C} \\
\Omega \in \mathcal{C} \\
\langle \langle \rangle, t[A] \rangle \cdot T \in \mathcal{C} \\
(t \neq f \cdot \forall(A', t'))
\end{array}
\]

**Morphisms**

\[
\begin{array}{c}
f \in \mathcal{C} \\
f \cdot \text{Cur}(A, t) \in \mathcal{C} \\
f \cdot \forall(A, t) \in \mathcal{C} \\
f \cdot \text{Snd} \in \mathcal{C}
\end{array}
\]

\[
\text{Fst}^k \cdot \text{Snd} : \langle \text{Id}, t_1 \rangle \cdot \text{App}; \cdots ; \langle \text{Id}, t_n \rangle \cdot \text{App} \in \mathcal{C}
\]

- Evaluating the environment \( f \) independently of the morphism \( t \) in an expression \( f \cdot t \) or not. The first choice is appropriate for an eager strategy and the second for a lazy one because in the lazy case \( t \) determines if an evaluation of a component of \( f \) to a value is necessary or not.
- Evaluating \( t_1 \) or \( t_2 \) first in an expression \( t_1 ; t_2 \).

The third factor is independent of the other two, and a choice may lead to some optimizations, but not to principal differences. If the first choice is adopted for an eager strategy, we get a strategy \( =>_E \), which yields an abstract machine that can be transformed directly into the CAM. The other choice together with a lazy strategy leads to a strategy \( =>_L \) that is the basis for another abstract machine close to Krivine's machine. The canonical morphisms for these reduction strategies are those morphisms \( f \cdot t \) where \( t \) corresponds to a weak head normal form in the \( \lambda \)-calculus and \( f \) is a canonical context morphism: for a precise definition in the lazy case see Table 3. The main difference in the eager case is the clause for environments, which is

\[
\text{Fst}^k; \langle f, t[A] \rangle \in \mathcal{C}
\]

The reduction to a combinator corresponding to the translation of a normal form in the \( \lambda \)-calculus can be done by a repeated reduction of the components of an abstraction to weak head normal form, i.e. by a repeated call to the previously described abstract machines. This explains why the definition of a canonical combinator contains a combinator \( \text{Fst}^k; \langle \langle \rangle, t_1, \ldots, t_n \rangle \): we want to ensure that \( f \) canonical implies \( \langle \text{Fst}; f, \text{Snd} \rangle \) canonical.
A later rule applies only if all prior rules fail. $f$ denotes always a canonical context morphism. The same conventions as in Table 3 apply.

### Context morphisms

\[
\begin{align*}
\frac{f;g \Rightarrow_L f';h \Rightarrow_L h'}{f;g(h) \Rightarrow_L f'(h')} \quad 
\frac{f;\langle \rangle \Rightarrow_L \langle \rangle}{f;\text{id} \Rightarrow_L f} \\
\frac{f';g, [A] \in \mathcal{G}}{f';g \Rightarrow_L h} \\
\frac{f;\text{Fst} \Rightarrow_L h}{f;\text{Id} \Rightarrow_L f} \\
\end{align*}
\]

### Types

\[
\begin{align*}
\frac{f=((h;g)*A) \Rightarrow_L B}{f(h;g)*A \Rightarrow_L B} \\
\frac{f*(h*(g*A)) \Rightarrow_L B}{f*(g*A) \Rightarrow_L B} \\
\frac{f*(g*A) \Rightarrow_L B}{f*T \Rightarrow_L h*\Pi(A,\langle \langle \rangle, t[\Omega] \rangle)*T} \quad 
\frac{f*Snd \Rightarrow_L h*\forall(A, t)}{f*Snd \Rightarrow_L s} \\
\frac{f*(g*A) \Rightarrow_L B}{f*(t;g*\text{Id}) \Rightarrow s} \\
\frac{f;\langle g, t[A] \rangle \in \mathcal{G}}{f;\text{Snd} \Rightarrow_L s} \\
\frac{f*(t;g*A) \Rightarrow_L B}{f*(t;g*A) \Rightarrow_L B} \\
\frac{f;\langle g, t[A] \rangle \in \mathcal{G}}{f;\text{App} \Rightarrow_L t} \\
\frac{f*(g*\text{App}) \Rightarrow_L h*\forall(A, t'* \text{Cur}(A, t'))}{f*(g*\text{App}) \Rightarrow_L h*\forall(A, t')} \\
\frac{f*(g*\text{App}) \Rightarrow_L h*\forall(A, t')}{f*(g*\text{App}) \Rightarrow_L h*\forall(A, t')} \\
\frac{f*(h*t;h*t') \Rightarrow_L s}{f*(g*t) \Rightarrow_L s} \\
\frac{f*(h*(t;t')) \Rightarrow_L s}{f*(g*t) \Rightarrow_L s} \\
\frac{f*(h*t;h*t') \Rightarrow_L s}{f*(g*t) \Rightarrow_L s} \\
\frac{f*(h*t;h*t') \Rightarrow_L s}{f*(g*t) \Rightarrow_L s} \\
\frac{f*(h*t;h*t') \Rightarrow_L s}{f*(g*t) \Rightarrow_L s} \\
\end{align*}
\]

The difference between the reduction strategies becomes apparent when we look at the way the access to an environment and the application are handled. The strategy $\Rightarrow_E$ can directly return the appropriate component of the environment in the first case whereas the lazy strategy has to schedule a reduction of the component as well. In the case of a combinator $t = f*(t_1; \langle \text{id}, t_2 \rangle \ast \text{App})$, the eager strategy reduces $f*t_1$ to a combinator $s$ and the combinator $f*t_2$ to $s_2$. If $s \neq h \ast \text{Cur}(A, t_1)$, the result is $(h \ast s); \langle \text{id}, s_2 \rangle \ast \text{App}$, otherwise the combinator $\langle \text{id}, s_2 \rangle \ast \text{App}$ is reduced to produce the result. The lazy strategy reduces only $f*t_1$ to determine if a combinator $h \ast \text{Cur}(A, t_1)$ results and if it does reduces the combinator $\langle h, t_2 \rangle \ast s_1$. All inference rules for the lazy case are listed in Table 4. The difference between the eager and the
The lazy strategy is captured by the rule for application, which for the eager case is as follows:

\[
(f \ast t) \Rightarrow_{E} h \ast \text{Cur}(A_1, s_1) \quad f; g \ast \text{Snd} \Rightarrow_{E} s_2 <h, s_2[A_1]> \ast s_1 \Rightarrow_{E} s
\]

\[
f \ast (t; g \ast \text{App}) \Rightarrow_{E} s
\]

The proof that both strategies describe for every combinator \( f \ast t \) with \( f \) canonical a reduction to canonical form uses an induction over the length of the longest \( \rightarrow_{N} \)-reduction sequence of \( f \ast t \); for details see Section 3.

### 2.4. Machines

The reduction rules describe how a combinator \( f; g \) or \( f \ast t \) is reduced where \( f \) is canonical. Both machines have therefore a environment register containing the machine representation \([f]_m\) of a context morphism \( f \), a code register containing \([g]_m\) or \([t]_m\) respectively and a normal form register containing the code of the canonical combinator to which \( f \ast t \) reduces. We will explain only the lazy machine here; the eager machine can be developed along similar lines. In an implementation, the combinators are represented as graphs, and the machine acts by walking through the left spine of the graph in the code register. Sharing in the lazy machine is implemented in the standard way by using several pointers to a common subexpression. We present this graph in the sequel in a linearized form. The identity combinator is represented by the empty code, which is denoted by \( \_ \). This code acts like an empty word, i.e. its concentration with another code sequence \( c \) is identical to \( c \). Otherwise, the code \([c]_m\) of a combinator \( c \) is identical to \( c \) except in the following cases:

\[
[f; g]_m = [f]_m [g]_m \quad [f \ast A]_m = [A]_m [f]_m
\]

\[
[f \ast t]_m = [t]_m [f]_m \quad [t, s]_m = [s]_m [t]_m
\]

The idea behind the transitions for the machines is that every inference rule

\[
f_1 \ast e_1 \Rightarrow e_1' \ldots f_n \ast e_n \Rightarrow e_n' \Rightarrow f \ast e \Rightarrow e'
\]

gives rise to machine transitions describing how \( e' \) can be constructed from the canonical combinators \( e_1', \ldots, e_n' \). Hence an induction over the definition of \( \Rightarrow_{L} \) and \( \Rightarrow_{E} \) suffices to show that both machines perform the reduction to the corresponding canonical combinators. For the definition see Table 5, and for the correctness proof see Theorem 17.

We discuss here only the transitions modelling access to environments and \( \beta \)-reduction in the lazy case. The inference rules are

\[
f; \langle g, t \rangle \in \mathcal{E} \quad f \ast t \Rightarrow_{L} s
\]

\[
f; \langle g, t \rangle \ast \text{Snd} \Rightarrow_{L} s
\]

\[
f \ast t' \Rightarrow_{L} \langle \text{id}, s' \rangle \ast \text{App} \quad f \ast t \Rightarrow_{L} h \ast \text{Cur}(A, t'') \quad <h, s'> \ast t'' \Rightarrow_{L} s
\]

\[
f \ast (t; t') \Rightarrow_{L} s
\]
The first is captured by a transition

\[
\begin{array}{c|c|c|c|c|c|c}
E & C & CF & E & C & CF \\
\hline
f \langle g, t \rangle & \text{Snd} & C & N & f & tC & N \\
\end{array}
\]

Table 5
Transitions of the lazy machine

A later rule applies only if all earlier ones fail. If we write \( e C \) for the content of the code register, we always assume that \( e \) is not a sequence of instructions. The rules marked with \((+)\) apply only if the flag for \( t \ast h \) and \( \langle h, t \rangle?(A) \) respectively indicate that \( h \) is the code for a canonical combinator. The rules for types are used only for the calculus of constructions. The code \( \langle f, t \rangle?(A) \) corresponds to the combinator \( \langle f, t[A] \rangle \). Therefore we omit the combinator \( ?(A) \) to obtain a machine for constant D-categories.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Code</th>
<th>Canonical form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \rangle )</td>
<td>( f )</td>
<td>( \langle \rangle C )</td>
</tr>
<tr>
<td>( \langle \rangle )</td>
<td>( C )</td>
<td>( C )</td>
</tr>
<tr>
<td>( (+)\text{Fst} )</td>
<td>( \text{Fst} \langle h, t \rangle?(A) )</td>
<td>( \text{Fst} C )</td>
</tr>
<tr>
<td>( \text{Fst} )</td>
<td>( f \langle g, t \rangle?(A) )</td>
<td>( \text{Fst} C )</td>
</tr>
<tr>
<td>( f )</td>
<td>( gC )</td>
<td>( gC )</td>
</tr>
<tr>
<td>( fg )</td>
<td>( C )</td>
<td>( C )</td>
</tr>
<tr>
<td>( * )</td>
<td>( f )</td>
<td>( (A \ast g) \ast hC )</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>( f )</td>
<td>( A \ast (h g)C )</td>
</tr>
<tr>
<td>( f )</td>
<td>( \Omega \ast gC )</td>
<td>( \Omega \ast gC )</td>
</tr>
<tr>
<td>( f )</td>
<td>( C )</td>
<td>( C )</td>
</tr>
<tr>
<td>( A \ast g )</td>
<td>( f )</td>
<td>( A \ast gC )</td>
</tr>
<tr>
<td>( f )</td>
<td>( gAC )</td>
<td>( gAC )</td>
</tr>
<tr>
<td>( T )</td>
<td>( f )</td>
<td>( TC )</td>
</tr>
<tr>
<td>( \text{Snd} )</td>
<td>( f )</td>
<td>( \text{Snd} TC C )</td>
</tr>
<tr>
<td>( T )</td>
<td>( f )</td>
<td>( TC C )</td>
</tr>
<tr>
<td>( T )</td>
<td>( f )</td>
<td>( C )</td>
</tr>
<tr>
<td>( A )</td>
<td>( f )</td>
<td>( AC )</td>
</tr>
<tr>
<td>( f )</td>
<td>( C )</td>
<td>( C )</td>
</tr>
<tr>
<td>( * )</td>
<td>( f )</td>
<td>( (t \ast g) \ast hC )</td>
</tr>
<tr>
<td>( f )</td>
<td>( t \ast (h g)C )</td>
<td>( N )</td>
</tr>
<tr>
<td>( - \ast g )</td>
<td>( f )</td>
<td>( - \ast gC )</td>
</tr>
<tr>
<td>( f )</td>
<td>( C )</td>
<td>( C )</td>
</tr>
<tr>
<td>( (+)\text{Snd} )</td>
<td>( f )</td>
<td>( \text{Fst} h )</td>
</tr>
<tr>
<td>( \text{Snd} )</td>
<td>( f \langle g, t \rangle?(A) )</td>
<td>( \text{Snd} C )</td>
</tr>
<tr>
<td>( \text{Snd} )</td>
<td>( f )</td>
<td>( tC )</td>
</tr>
<tr>
<td>( \text{Snd} )</td>
<td>( C )</td>
<td>( C )</td>
</tr>
</tbody>
</table>
Table 5 (continued)
Transitions of the lazy reduction machine

<table>
<thead>
<tr>
<th>Context</th>
<th>Code</th>
<th>Canonical form</th>
</tr>
</thead>
<tbody>
<tr>
<td>(β)</td>
<td>f</td>
<td>N App * (¬, s)</td>
</tr>
<tr>
<td></td>
<td>⟨f, s⟩? (A)</td>
<td></td>
</tr>
<tr>
<td>App</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>App * ⟨¬, t⟩ C</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>N App * ⟨¬, t ∗ f⟩ a</td>
</tr>
<tr>
<td>App</td>
<td>f</td>
<td>App ∗ g C</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>N App * ⟨¬, (Snd ∗ g) ∗ f⟩ a</td>
</tr>
<tr>
<td>h ∗ ⟨t, t′⟩</td>
<td>f</td>
<td>(t ∗ t′) ∗ h C</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>t ∗ h t′ ∗ h C</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>h t C</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

The flag indicating that f is canonical is set as well.

which schedules the evaluation of the appropriate component of the environment. The register names E, C and CF stand for environment, code and canonical form respectively. The transitions for the second rule use the fact that any derivation tree of $f ∗ t′ \Rightarrow L h ∗ \text{Cur}(A, t'')$ has a branch with a leaf

$$h ∗ \text{Cur}(A, t'') \Rightarrow L h ∗ \text{Cur}(A, t'')$$

Hence any transition sequence for the code for t passes through a state

<table>
<thead>
<tr>
<th>E</th>
<th>C</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>[h]m</td>
<td>Cur([A]m, [t'']m) C</td>
<td>N</td>
</tr>
</tbody>
</table>

Therefore it suffices to introduce the transition

<table>
<thead>
<tr>
<th>E</th>
<th>C</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Cur(A, t) C</td>
<td>N App * ⟨ld, t'⟩</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t C</td>
</tr>
</tbody>
</table>

The access to the environment admits an important optimization. Consider the combinator $f ∗ \text{Cur}(A, \text{Snd}) ; ⟨ld, t⟩ ∗ \text{App}$. According to the strategy $⇒_L$, its reduction amounts to reducing the combinator $⟨f, t[A]⟩ ∗ \text{Snd}$, which in turn leads to the reduction of the combinator $\text{ld} ∗ (f ∗ t)$. Because f is canonical, the derivation tree for the latter combinator contains the judgement $f ∗ t \Rightarrow L s$. So we obtain an admissible inference rule

$$f ∈ G \quad \text{Fst}^k; f ∗ t \Rightarrow L s$$

$$\frac{\text{Fst}^k; ⟨g, f ∗ t⟩ ∗ \text{Snd} ⇒ L s}{\text{Fst}^k; ⟨g, f ∗ t⟩}$$
In the same way we get another admissible inference rule

\[
\frac{\phi \in \Phi}{Fst^\phi \langle f, t \rangle; Fst \Rightarrow L Fst^\phi \cdot \phi}
\]

The definition of machine transitions for these inference rules depends on recognizing that certain context morphisms are canonical. This is easily achieved by introducing two flags that indicate whether in a combinator \( Fst^\phi \langle f, t \rangle \) and \( f \cdot t \) respectively \( f \) is canonical or not. This marking procedure is also used for avoiding the reevaluation of a component of an environment that is already canonical.

Both abstract machines can be implemented easily in a language like C, which gives the programmer access to many details of the architecture of the underlying computer, as well as in a language like ML, which is a functional programming language with a very powerful type discipline. In both cases, the combinators \( Fst^\phi \) and \( Fst^\phi \cdot \text{Snd} \) are represented by an integer, and the combinator \( \langle f, t \rangle \) by a list. In particular, the combinator \( \langle \cdot \rangle \) plays the role of the empty list. The machine instructions can be easily described in terms of these lists. The pattern matching in ML simplifies the implementation of the machines enormously. The reason is that the ML-code for the machine transitions is essentially a big case-statement where every alternative represents one machine transition. Hence the abstract machines are well-suited for theorem provers written in ML like LEGO and Coq.

If we restrict the lazy machine to the code for the combinators occurring during the reduction of combinators that are transitions of \( \lambda \)-expressions, we obtain Krivine's machine. The \( \lambda \sigma \)-calculus has been used to derive extensions of Krivine's machine for reduction to normal form [4]. The calculus adds explicit substitution on top of the \( \lambda \)-calculus. In our notation, the typed version has the raw expressions

\[
\begin{align*}
\Gamma & ::= \emptyset \mid \Gamma \cdot A \\
f & ::= \langle \cdot \rangle \mid \text{Id} \mid f \cdot f \mid \text{Fst} \langle \cdot \rangle \\
A & ::= G \mid \Pi(A, A) \\
t & ::= \text{Snd} \mid \lambda A.t \mid tt \mid f \cdot t
\end{align*}
\]

An expression \( f \) is usually called a substitution. The typing rules of the raw expressions are those of simply-typed \( \lambda \)-calculus with the following rules for the extra expressions:

\[
\begin{align*}
\Gamma \vdash \langle \cdot \rangle : \emptyset \\
\Gamma \vdash \text{Id} : \Gamma \\
\Gamma \vdash \langle f, t \rangle : A \cdot A \\
\Gamma \vdash f : A \quad \Gamma \vdash t : A \\
\Gamma \vdash g : \Gamma'' \\
\Gamma \vdash f \cdot g : \Gamma' \\
\Gamma' \vdash f : \Gamma \\
\Gamma' \vdash g : \Gamma''
\end{align*}
\]

This calculus becomes a subsystem of the combinators for constant \( D \)-categories if we replace \( ts \) by \( t \cdot \langle \text{Id}, s \rangle \cdot \text{App} \) and the judgement \( \Gamma \vdash t : A \) by \( \Gamma \cdot t : 1 \to A \). Therefore the \( D \)-categories are an appropriate categorial framework for the \( \lambda \sigma \)-calculus.
An extension of Krivine's machine to a machine for the reduction of a term \( f \ast t \) to a normal form is given in [1] as well. It does not use the notion of a canonical environment \( f \) and hence does not reduce environments to a canonical form but lists all possible cases for the reduction of \( f \ast (g \ast t) \), where the code for \( f \) is stored in the environment register and the code for \( g \ast t \) in the code register. Hence the access to an environment \( \langle \text{Fst}; f, t \rangle \) is more complicated than in the machine presented here.

Crégut's machine [4] is based on the variant of the \( \lambda\sigma \)-calculus given in [5] in connection with multicategories. This variant replaces the substitutions \( \text{Fst} \) and \( \text{Id} \) by the lists \((n-1, \ldots, 1)\) and \((n-1, \ldots, 0)\), where \( n \) is the number of free variables of the expression to which this substitution is applied. Hence all expressions have to be decorated with the number of free variables in order to formulate the reduction rule corresponding to \( f \ast \text{Cur}(A, t) \rightarrow^{N} \text{Cur}(A, \langle \text{Fst}; f, \text{Snd}[A] \rangle \ast t) \). Furthermore the substitution cannot be stored directly in the environment register because otherwise the transformation from the substitution corresponding to a combinator \( f \) to that corresponding to the combinator \( \langle \text{Fst}; f, \text{Snd} \rangle \) becomes prohibitively expensive. Crégut's machine avoids such a complicated manipulation by handling the free variables that occur during the reduction inside a \( \lambda \)-abstraction differently from the other variables. For this purpose the machine maintains an index that is increased every time we start a reduction inside a \( \lambda \)-abstraction. The value of this index is stored as the value of such a free variable. This is a kind of reverse de Bruijn-numbering, where the origin is the root of the term. The translation from this reverse index to the normal de Bruijn-index is done whenever the value of such a variable is computed. The special role of these free variables has no counterpart in the \( \lambda\sigma \)-calculus, and so the correspondence between the calculus and the machine is based on a translation of the reverse indexing into standard de Bruijn-numbers.

It is no surprise that the eager machine does not directly specialize to the CAM because the latter is based on a different categorical structure, namely cartesian closed categories. But if we consider only the combinators occurring during the reduction of translations of \( \lambda \)-expressions and identify the canonical form- and environment-register, we obtain the CAM.

3. Extensions for the calculus of constructions

Most of the changes that transform the machines for the typed \( \lambda \)-calculus into those for the calculus of constructions are due to the dependent types in the latter calculus.

3.1. The calculus of constructions

Calculi with dependent types have a nontrivial equality relation on types. As a consequence, properties like subject reduction or uniqueness of typing, which are obvious in the simply typed \( \lambda \)-calculus, cannot be taken for granted in those calculi. As these properties are essential for implementations, an important step towards the
The design of abstract machines is to show that the calculus of constructions satisfies them. There are two different approaches to ensure these properties:

1. Extra type information is added, i.e. the term \( ts \) is replaced by \( \text{App}(x.A,B,t,s) \), and equations are defined between terms of the same type. In this way subject reduction and uniqueness of typing are ensured by the formalism.

2. Reduction is defined on the level of raw terms, and confluence implies both properties.

The first version is used for establishing the correspondence between category theory and the calculus, and the second one is the basis for implementations because an inspection of the structure of a term suffices to check whether a reduction rule applies. The equivalence of these two versions is based on the confluence of the reduction in the implicit calculus, which implies that \( \Pi x:A.B = \Pi x:A'.B' \) holds iff \( A=A' \) and \( B=B' \).

We will use therefore the second version of the calculus of constructions. The raw expressions are given by the grammar

\[
\begin{align*}
\Gamma &::= [ ] \mid (\Gamma, x:E) \\
E &::= \Pi x:E.E \mid | \text{Prop} \mid \text{Proof}(t) \\
t &::= x \mid \lambda x:E.t \mid tt \mid \forall x:E.t \\
\end{align*}
\]

where \( \Gamma \) stands for contexts, \( E \) for types and \( t \) for terms. The rules for valid judgments are listed in Table 6. There are three reduction rules for raw expressions:

- (\( \beta \)-rule) \( (\lambda x:A.t)s \mapsto t[x/s] \)
- (\( \forall \)-elim) \( \text{Proof}(\forall x:A.p) \mapsto \Pi x:A.\text{Proof}(p) \)
The third rule, called coherence rule, characterizes a proof of a universally quantified proposition $\forall x: A. p$. It is a dependent function that, given any $x$ of type $A$, constructs a proof of $p(x)$.

### 3.2. The categorical structure

The categorical structure used for modelling the calculus of constructions [9] is a generalization of the structure discussed in the last section. The most important point is the removal of the restriction that there is a set of types which is the set of objects in all fibres. Furthermore we add an object $\Omega$ in $E(\cdot \Omega)$ corresponding to the type $\text{Prop}$ and an object $T$ in $E(\cdot [\cdot \Omega])$ representing $\text{Prop}(x)$ in the contest $x: \text{Prop}$. The universal quantification is captured by the conditions

- For every morphism $t : 1 \rightarrow \langle \rangle^*(\Omega)$ in $E(\Gamma \cdot A)$ there exists a morphism $\forall(A, t) : 1 \rightarrow \langle \rangle^*(\Omega)$ in $E(\Gamma)$ and the naturality condition

  $$h^* \forall(A, t) = \forall(h^*A, (h \cdot \text{id})^*t)$$

  holds for every $h : \Gamma' \rightarrow \Gamma$.

- For every object $A$ in $E(\Gamma)$ and any morphism $t : 1 \rightarrow \Omega$ in $E(\Gamma \cdot A)$, the following coherence condition holds:

  $$\langle \langle \rangle, \forall(A, t) \rangle^*(T) = \Pi(A, \langle \langle \rangle, t \rangle^*(T))$$

### 3.3. The combinators

The ideas behind the derivation of an equational presentation for the constant D-categories apply to the more general case as well. However, the dependent types make the definition of a well-formed combinator more complex. First, properties of the combinators can only be established by an induction over the derivation of combinators if type information is added to the combinators $\langle f, t \rangle$ and $\text{App}$, i.e. they are replaced by $\langle f, t[A] \rangle$ and $\text{App}(A, B)$. Second, the weakening that could be suppressed in the rules for the combinators $\text{Snd}$ and $\text{App}$ now becomes apparent with $F \cdot A, Snd : 1 \rightarrow \text{Fst} \cdot A$ and $\Gamma \cdot A, \text{App}(A, B) : \text{Fst} \cdot \Pi(A, B) \rightarrow B$ because the type $A$ is well-formed only in context $\Gamma$. The equivalence between the calculus and the combinators can be shown by an induction over the derivation. We will discuss here only the implicit version on which the abstract machines are based; for further details see [23]. This version replaces the combinator $\text{App}(A, B)$ by $\text{App}$ and defines reduction on raw combinators. The relation between implicit and explicit combinators is the same as the relation between the explicit and the implicit calculus. Especially, the implicit combinators are a convenient shortcut for the explicit ones because one can show by using the confluence that every implicit combinator can be uniquely extended to an explicit one.
Now we turn to the proof of the equivalence between the implicit calculus and the implicit combinators and of confluence and subject reduction. We start by extending the definition of the translations \([-\] and \((-)^c\) as follows:

(1) Translation \([-\] from the calculus into combinators:

\[
\begin{align*}
\text{Prop} & = \langle \rangle \ast \Omega \\
\text{Proof}(t) & = \langle \langle \rangle, \lfloor t \rfloor \rfloor \ast T \\
\forall A.t & = \forall (\lfloor A \rfloor, \lfloor t \rfloor)
\end{align*}
\]

(2) Translation \((-)^c\) from the combinators back into the calculus:

\[
\begin{align*}
\langle -, - \rangle & \quad (n, \langle f, t[A] \rangle)^c = \{(n, f)^c, (n, t)^c\} \\
(*) & \quad \frac{(n, f)^c = \{t_{m-1}, \ldots, t_0\}}{(n, f \ast A)^c = (m, A)^c[x_i \backslash t_i]} \\
(\Omega) & \quad (n, \Omega)^c = \text{Prop} \\
(T) & \quad (n, T)^c = \text{Proof}(x_0) \\
(\forall) & \quad (n, \forall(A, t)^c = \forall(n, A)^c.(n + 1, t)^c
\end{align*}
\]

The soundness of the translation \([-\] can be shown in a standard way:

**Proposition 5.** Let \(e\) and \(e'\) be any CC-expression and \(\Gamma\) be any well-formed context.

(i) If \(e \rightsquigarrow e'\), then also \([e] \rightsquigarrow [e']\).

(ii) For every judgement in the calculus there exists a corresponding judgement in the equational theory of combinators, more precisely:

\[
\begin{align*}
\text{(Contexts)} & \quad \vdash \Gamma \text{ ctxt implies } [\Gamma] \in \text{Obj} \\
\text{(Types)} & \quad \Gamma \vdash A \text{ type implies } [\Gamma] \triangleright [A] \\
\text{(Terms)} & \quad \Gamma \vdash t : A \text{ implies } [\Gamma] \triangleright [t] : 1 \rightarrow [A]
\end{align*}
\]

**Proof.** The proof depends crucially on the fact that weakening and substitution, which are meta-operations in the theory, are translated into certain categorical combinators in the equational theory. Their proofs are routine inductions over the structure of raw CC-expressions and are therefore omitted here.

**Lemma 6.** Weakening of an expression \(e\) well-formed in context \((\Gamma, \Gamma')\) with \(\Gamma' = B_{i-1} \cdots B_0\) with respect to types \(A_{m-1}, \ldots, A_0\) corresponds to the application of the combinator

\[
S^n := \text{Fst}^m \cdot \text{Id}[B_{i-1}] \cdots \text{Id}[B_0],
\]

where \(i\)-times
More precisely, if $U^n_t(e)$ denotes the expression $e$ where all variables $x_j$ are replaced by $x_{j+m}$ for $j \geq i$ and if we define $S^n_t(\Gamma)$ as an abbreviation for

\[
S^n_t([\ []]) = [\ ] \\
S^n_t(\Gamma \cdot A) = S^n_t(\Gamma) \cdot S^n_t(A) \quad (i > 0) \\
S^n_t(\Gamma) = \Gamma \quad (i = 0),
\]

then for a given context $(\Gamma', \Gamma')$ with $\Gamma' = B_{i-1} \cdots B_0$ and types $A_{n-1}, \ldots, A_0$ we have with $A := (\Gamma, A_{n-1}, \ldots, A_0, U^n_t(\Gamma'))$ for any raw type $A$ and term $t$

(i) $[\Gamma] \cdot [A_{n-1}] \cdots [A_0] \cdot S^n_t([\Gamma']) \rightarrow^* [A]$

(ii) $S^n_t([A]) \rightarrow^* [U^n_t(A)]$

(iii) $S^n_t([t]) \rightarrow^* [U^n_t(t)] : 1 \rightarrow [U^n_t(A)]$

The next lemma is concerned with substitution, which is also modelled by an operator in the combinators.

**Lemma 7.** Substitution of a term $s$ of type $A$ in an expression $e$ well-formed in a context $(\Gamma, \Gamma')$, where $\Gamma' = B_{n-1} \cdots B_0$, for the variable $x_n$ corresponds to the application of the combinator

\[
S^u_t([s]) := \langle Id, [s] [\ ] \rangle \cdot \langle Id, \ldots, Id \rangle \cdot \ldots \cdot \langle Id, [s] \rangle \cdot \langle Id, t \rangle \cdot [A]
\]

More precisely, if we define $S^u_t([s])((\Gamma))$ as an abbreviation for

\[
S^u_t([s])([\ ]) = [\ ] \\
S^u_t([s])(\Gamma \cdot E) = S^u_t(\Gamma) \cdot S^u_t([s]) \cdot E \quad (n > 0) \\
S^u_t([s])(\Gamma) = \Gamma \quad (n = 0)
\]

then for a given context $(\Gamma, A, \Gamma')$ and a term $s$, we have with $A := (\Gamma, \Gamma'[x_n \backslash s])$

(i) $[\Gamma] \cdot S^u_t([s])((\Gamma')) \rightarrow^* [A]$

(ii) $S^u_t([s]) \cdot [B] \rightarrow^* [B[x_n \backslash s]]$

(iii) $S^u_t([s]) \cdot [t] \rightarrow^* [t[x_n \backslash s]]$

Now we can show the main proposition.

(i) We verify only the $\beta$-rule because it is the most interesting one. By definition, $[[\lambda x : A.t]s] = \text{Cur}([A], [t]); \langle Id, [s] \rangle \cdot \text{App}$, and the substitution lemma yields

\[
\text{Cur}([A], [t]); \langle Id, [s] \rangle \cdot \text{App} \rightarrow^* \langle Id, [s] \rangle \cdot [t[x \backslash s]]
\]
(ii) Again, we verify only the application. So let us assume we have \( \Gamma \vdash \mathit{t} : \Pi X : A.B \), \( \Gamma \vdash \mathit{s} : A' \) and \( A \leftrightarrow A' \). The induction hypothesis yields \( \llbracket \Gamma \rrbracket \gg [\mathit{t}] : 1 \rightarrow \Pi (\llbracket A \rrbracket, \llbracket B \rrbracket) \) and \( \llbracket \Gamma \rrbracket \gg [\mathit{s}] : 1 \rightarrow \llbracket A' \rrbracket \). By part (i), \( \llbracket A \rrbracket \leftrightarrow \llbracket A' \rrbracket \), and so we have

\[
\llbracket \Gamma \rrbracket \gg [\mathit{t}] ; \langle \text{Id}, [\mathit{s}] \rangle \ast \text{App} : 1 \rightarrow \langle \text{Id}, [\mathit{s}] \rangle \ast \llbracket B \rrbracket
\]

This completes the proof of Proposition 5. \( \square \)

As already mentioned, the confluence of the combinators, subject reduction and the soundness of the translation \((-c\) have to be proved together. We begin with reduction of raw combinators:

**Proposition 8.** Let \( e \) and \( e' \) be any raw combinators such that \( e \rightarrow e' \). Then for any natural number \( m \)

\[
(m, e)^c \rightarrow^* (m, e')^c
\]

**Proof.** Similar to Proposition 5. \( \square \)

The next lemma makes it possible to use the confluence of the calculus to show the combinators to be confluent as well.

**Lemma 9.** For any type or morphism in the fibre \( e \)

\[
e \rightarrow^* \llbracket (m, e)^c \rrbracket
\]

**Proof.** Induction over the structure of \( e \). \( \square \)

Now we can show the confluence.

**Theorem 10.** The reduction \( \rightarrow \) on combinators is confluent for types and morphisms.

**Proof.** Consider any types or morphisms \( e \) and \( e' \) such that \( e \leftrightarrow e' \). By Theorem 8, we have \( (m, e)^c \leftrightarrow^* (m, e')^c \), and so the confluence of the calculus of constructions implies the existence of a combinator \( d' \) such that

\[
(m, e)^c \rightarrow^* d', \quad (m, e')^c \rightarrow d'
\]

Theorem 5 yields now

\[
\llbracket (m, e)^c \rrbracket \rightarrow^* \llbracket d' \rrbracket, \quad \llbracket (m, e')^c \rrbracket \rightarrow \llbracket d' \rrbracket
\]

Hence the previous lemma yields the claim. \( \square \)

We obtain subject reduction and uniqueness of typing for the combinators as corollaries.
Corollary 11. Let $\Gamma$ be any well-formed context and $f, A$ and $t$ be any combinators. Then we have

(i) $\Gamma \rightarrow \Delta$ implies $\Delta$ is well-formed.
(ii) $\Gamma \vdash f : \Delta$ and $f \rightarrow f'$ implies $\Gamma \vdash f' : \Delta$.
(iii) $\Gamma \vdash A$ and $A \rightarrow A'$ implies $\Gamma \vdash A'$.
(iv) $\Gamma \vdash t : 1 \rightarrow A$ and $t \rightarrow t'$ implies $\Gamma \vdash t' : 1 \rightarrow A$.

Proof. Induction over the definition of $\rightarrow$. The important case is the $\beta$-rule $f \ast \text{Cur}(A, t); g \ast \text{App} \rightarrow \langle f, g \ast \text{Snd}[A] \rangle \ast t$. The well-formedness of the left-hand side implies $\Gamma \vdash f : \Delta$, $\Delta \cdot A \vdash t : 1 \rightarrow B$, $\Gamma \vdash g : \Delta' \cdot A'$ and $\Delta \cdot \text{App} : \text{Fst} \ast \Pi(A, B')$ and the existence of a type $C$ such that $C \leftrightarrow \Pi(A, B)$ and $C \leftrightarrow g; \text{Fst} \ast \Pi(A', B')$. The confluence implies that

$$f \ast A \leftrightarrow g; \text{Fst} \ast A'$$

and so $\Gamma \vdash \langle f, g \ast \text{Snd}[A] \rangle \ast t : 1 \rightarrow \langle f, g \ast \text{Snd}[A] \rangle \ast B$. □

The uniqueness of typing follows also from the confluence:

Corollary 12. For all well-formed context $\Gamma$ and all raw combinators $f, A$ and $t$:

(i) If $\Gamma \vdash f : \Delta$ and $\Gamma \vdash f : \Delta'$, then $\Delta \leftrightarrow \Delta'$.
(ii) If $\Gamma \vdash t : 1 \rightarrow A$ and $\Gamma \vdash t : 1 \rightarrow A'$, then $A \leftrightarrow A'$.

Proof. Induction over the sum of the length of the derivation of $\Gamma \vdash f : \Delta$ and $\Gamma \vdash f : \Delta'$ and the sum of $\Gamma \vdash t : 1 \rightarrow A$ and $\Gamma \vdash t : 1 \rightarrow A'$ respectively. □

The soundness of the translation $(-)^f$ is an easy consequence of the last two corollaries:

Theorem 13. The inverse translation respects the judgements, more precisely for any context $\Gamma$ with $|\Gamma| = n$ we have:

(Contexts) $\Gamma \in \text{Obj}$ implies $(n, \Gamma)^f \vdash \text{ctxt}$

(Context Morphisms) $\Gamma \vdash f : A = [\cdot] \cdot B_{m-1} \cdots B_0$ implies

$(n, \Gamma)^f \vdash (n, f)^f = \{t_{m-1}, \ldots, t_0\}$ and

$(n, \Gamma)^f \vdash (n, t_i)^f$:

$(m-i-1, B_i)^f[x_{m-2-i} \setminus t_{m-1}, \ldots, x_0 \setminus t_{i+1}]$

(Types) $\Gamma \vdash A$ implies $(n, \Gamma)^f \vdash (n, A)^f \vdash \text{Type}$

(Terms) $\Gamma \vdash t : 1 \rightarrow A$ implies $(n, \Gamma)^f \vdash (n, t)^f : (n, A)^f$
Proof. Induction over the definition of $\gg$. Again, the interesting case is the application $t:g\ast\text{App}$. As in the proof of Corollary 11, we have $\Gamma \vdash t:1 \rightarrow C$, $\Gamma \vdash g\ast\text{Snd}:1 \rightarrow A'$ with $C \leftarrow \ast \Pi(A,B)$ and $A \leftrightarrow A'$. Theorem 5 and the induction hypothesis imply therefore $(n,\Gamma)\gg (n,t)\gg (n,C)\gg (n,g\ast\text{Snd})\gg (n,A')\gg$ with $(n,C)\gg \Pi(n,A)\gg (n+1,B)\gg$ and $(n,A)\gg A'\gg$. Hence $(n,\Gamma)\gg (n,t;g\ast\text{App})\gg (n+1,B)\gg [X_0\setminus(n,g\ast\text{Snd})\gg]$. □

Finally we can prove that the translation $(\gg)$ and $[-\gg]$ are inverse to each other in the sense discussed above.

**Theorem 14.** The translations $(\gg)$ and $[-\gg]$ are inverse to each other in the following sense, where $n$ denotes the length of the context $\Gamma$:

(i) $([-\gg])\gg$ is the identity:
(1) on contexts: $(n,[\Gamma])\gg \equiv \Gamma$
(2) on types: $(n,[A])\gg \equiv A$
(3) on terms: $(n,[t])\gg \equiv t$

(ii) The combinator $e$ reduces to $([e])\gg$:
(1) on contexts: $\Gamma \gg \ast ([n,\Gamma])\gg$
(2) on context morphisms: $f;\text{Fst}\ast\text{Snd}\gg \ast ([t])\gg$ if $(n,f)\gg \equiv \{t_{m-1},\ldots,t_0\}$
(3) on types: $A \gg \ast ([n,A])\gg$
(4) on terms: $t \gg \ast ([n,t])\gg$

Proof. Induction over the structure of the raw combinators. □

As already mentioned in the previous section, there exists no proof as yet of the strong normalization of $\gg$. Therefore we introduce reductions $\gg$ and $\gg$ describing the reduction to a combinator without any outer $\beta$-redex and to normal form respectively. The reduction $\gg$ inherits all important properties from $\gg$, as the following theorem shows:

**Theorem 15.** (1) Both $\gg$ and $\gg$ have the same normal forms of types and morphisms, namely the translation of the normal forms of types and morphisms of the Calculus of Constructions.

(ii) $\gg$ is confluent on morphisms and types.

(iii) $e\leftrightarrow\ast e'\iff e\leftrightarrow\ast e'$ if $e$ is a morphism or a type.

Proof. (i) Induction over the structure of types and morphisms.

(ii) Consider any $\gg$-normal form $d$ of $e$ and $d'$ of $e'$. Because any $\gg$-reduction is also a $\gg$-reduction, the previous part and the confluence imply $d\equiv d'$.

(iii) Consequence of the previous part. □

3.4. Reduction strategies

It is not only necessary to add a new clause for the universal quantification to the reduction rules for morphisms, but the dependent types require also a reduction on
types. All the discussions of the previous section about the principles of eager and lazy reduction apply to the combinators for the calculus of constructions as well and yield the definition of canonical combinators in Table 3 and of the reduction strategies in Table 4. Now we show that both strategies describe a reduction to a canonical form.

**Theorem 16.** For both strategies $\Rightarrow_L$ and $\Rightarrow_E$, for every canonical context morphism $f$ and every well-formed combinator $g$, $B$ and $t$ with $\Gamma : t : 1 \rightarrow C$ such that $f ; g$, $f * B$ and $f * t$ are well-formed, there exist unique canonical combinators $h$, $A$ and $s$ such that

(i) $f ; g \Rightarrow h$

(ii) $f * B \Rightarrow A$

(iii) $f * t \Rightarrow s$

**Proof.** We first consider the proof for the lazy strategy $\Rightarrow_L$. By an induction over the structure of $e$ we show that the theorem is true if it holds for any $e'$ and $f'$ such that $v(f' * e') < v(f * e)$, where $v(d)$ is the length of the longest $\Rightarrow_N$-reduction sequence of $d$. We consider here only the case of a $\beta$-reduction.

$(t ; g * App)$ By induction hypothesis, $f * t \Rightarrow_L s$ with $s$ canonical. First consider the case where $s \equiv h * Cur(A, t')$. Because $\langle h, g * SND[A] \rangle * t' \equiv v(f * t ; g * App)$, the induction hypothesis yields the claim. If $s \equiv h * Cur(A, t)$, we get immediately $f * (t ; g * App) \Rightarrow_L s ; \langle ld, f ; g * SND \rangle * App$.

The proof for the eager case is similar:

$(t ; g * App)$ The induction hypothesis implies $f * t \Rightarrow_E s$ and $f ; g * SND \Rightarrow_E s'$ for canonical $s$ and $s'$. If $s \not\equiv h * Cur(A, t')$ then the combinator $s ; \langle ld, s' \rangle * App$ is canonical. Otherwise we have $v(\langle h, s' [A] \rangle * t') < v(t ; g * App)$, and therefore the induction hypothesis yields $\langle h, s' \rangle * A \Rightarrow_E s''$ with $s''$ canonical. $\Box$

**Remark.** The proof of the eager case explains why we have to use the strong normalization for the relation $\Rightarrow_N$ and cannot rely on the strong normalization of the calculus. If we use $v((e')^c)$ instead of $v(e)$ the above proof breaks down. Consider the case $f * t ; \langle ld, s \rangle * App$ and suppose the induction hypothesis yields $f * t \Rightarrow_E f * \langle Cur(A, g * t') \rangle * t'$ and $f * s \Rightarrow_E s'$. Then it is in general false that

$v(\langle f', g ; Frst^k * SND \rangle^c) < v(f * (t ; \langle ld, s \rangle * App))$

because the variable $k$ may not occur in $(t')^c$.

3.5. The abstract machines

The machines are also direct extensions of the ones described in the previous section. The additional reduction on types implies that the canonical form register may contain types and morphisms, and the inference rules suggest directly the transitions for the machines, which are listed in Table 5. We show here only the correctness of the lazy machine.
Theorem 17. For every canonical context morphism $f$ and all combinators $g$, $A$ and $t$ such that there exist combinators $h$, $B$ and $s$ satisfying $f; g \Rightarrow_L h$, $f \ast A \Rightarrow_L B$ and $f \ast t \Rightarrow_L s$ respectively, the machine performs the following actions:

$$
\begin{array}{|c|c|c|c|c|c|}
\hline
E & C & CF & E & C & CF \\
\hline
[f]_m & [g]_m C & N & [h]_m & C & N \\
\hline
[f]_m & [A]_m C & ? & C & [B]_m \\
\hline
[f]_m & [t]_m C & ? & C & [s]_m \\
\hline
\end{array}
$$

If $C$ is the empty code sequence, then the resulting states are final.

Proof. As already mentioned, the inference rules for the $\beta$-reduction makes it necessary to consider not only the transitions to a final state but also to previous ones if the code for a morphism is executed. Therefore we modify the statement of the theorem for morphisms as follows:

Theorem 17 (modified). If $f \ast t \Rightarrow_L s$, then we have the following cases, according to the structure of $s$:

1. $s \equiv \langle \text{Id}, s_1 \rangle \ast \text{App}; \langle \text{Id}, s_2 \rangle \ast \text{App}; \ldots$, and

$$
\begin{array}{|c|c|c|c|c|c|}
\hline
E & C & CF & E & C & CF \\
\hline
[f]_m & [t]_m C & N & [f]_m & C & N [s]_m \\
\hline
\end{array}
$$

2. $s \equiv d \ast u; v_1; \ldots; v_n$ and

$$
\begin{array}{|c|c|c|c|c|c|}
\hline
E & C & CF & E & C & CF \\
\hline
[f]_m & [t]_m C & N & [d]_m & [u]_m C & N [v_n]_m \cdots [v_1]_m \\
\hline
\end{array}
$$

3. $s \equiv \text{Snd}; v_1; \ldots; v_n$ and

$$
\begin{array}{|c|c|c|c|c|c|}
\hline
E & C & CF & E & C & CF \\
\hline
[f]_m & [t]_m C & N & \text{Snd} C & N [v_n]_m \cdots [v_1]_m \\
\hline
\end{array}
$$
To simplify the notation, we write all three cases as

$$\begin{array}{c|c|c|c|c|c|c}
  & \text{E} & \text{C} & \text{CF} & \text{E} & \text{C} & \text{CF} \\
\hline
[f]_m & [t]_m & C & N & d' & u' & C & Nv'
\end{array}$$

Theorem 16 implies that it is enough to show that the theorem holds for the conclusion of any inference rule for the strategy $\Rightarrow_L$ if it holds for all of its premises. We consider as an example the $\beta$-reduction. The machine sequence is

<table>
<thead>
<tr>
<th>Environment</th>
<th>Code</th>
<th>Canonical Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[f]_m$</td>
<td>$[t']_m[t]_m C$</td>
<td>$N$</td>
</tr>
<tr>
<td>I.H.</td>
<td></td>
<td>$\Downarrow_L^*$</td>
</tr>
<tr>
<td>$[f]_m$</td>
<td>$[t]_m$</td>
<td>$N \text{ App } \langle -, [s']_m \rangle$</td>
</tr>
<tr>
<td>I.H.</td>
<td></td>
<td>$\Downarrow_L^*$</td>
</tr>
<tr>
<td>$[h]_m$</td>
<td>$\text{Cur}([A]_m, [t'']_m) C$</td>
<td>$N \text{ App } \langle -, [s']_m \rangle$</td>
</tr>
<tr>
<td>I.H.</td>
<td></td>
<td>$\Downarrow_L$</td>
</tr>
<tr>
<td>$\langle [h]_m, \text{Snd} * [s']_m \rangle$</td>
<td>$[t'']_m C$</td>
<td>$N$</td>
</tr>
<tr>
<td>I.H.</td>
<td></td>
<td>$\Downarrow_L^*$</td>
</tr>
<tr>
<td>$d'$</td>
<td>$u' C$</td>
<td>$Nv'$</td>
</tr>
</tbody>
</table>

This completes the proof of Theorem 17. ☐

3.6. The type checking

Because of the dependent types the task of checking if a term has a given type may involve reductions. This task can be reduced to that of calculating a type $A$ such that $\Gamma \vdash t : A$, the so-called type synthesis, because if $A_1$ and $A_2$ are two types such that $\Gamma \vdash t : A_1$ and $\Gamma \vdash t : A_2$ then $A_1$ and $A_2$ are convertible, which is decidable. The inference rules for well-typed expressions do not specify an algorithm for type synthesis because they are not syntax-directed. The reason is
the conversion rule

\[ \Gamma \vdash t : A \quad A \leftrightarrow^* B \quad \Gamma \vdash B \text{ type} \]

\[ \Gamma \vdash t : B \]

which may be applied at any stage during the derivation of the well-formedness. Harper and Pollack [12] solve this problem by observing that the conversion rule is only necessary at certain stages during type checking and moreover can be replaced by two tests. The first determines if a type \( A \) is a dependent product or a dependent sum by a reduction of \( A \) to its weak head normal form (WHNF), which yields directly the outermost constructor, and the second is a test for convertibility.

This line of thought applies also to the type checking of combinators. It can likewise be reduced to type synthesis, and the conversion rule for well-typed combinators raises exactly the same problem. A solution consists as above of restricting the application of the conversion rule and replacing it with the above tests. Because the combinators have an explicit substitution operation it is not necessary to reduce combinators until the translation of a WHNF is reached, but we can stop at a combinator \( f \star B \) if we know that the outermost constructor of \( f \star B \) is that of \( B \). The canonical combinators are defined in such a way that on one hand \( f \) is as general as possible and on the other hand the above property of \( f \star B \) still holds. As we will see in a moment, the postponement of substitution captured by this definition is crucial for the efficiency of type synthesis. So both reduction machines described in the previous chapter basically implement the first test. The convertibility test is based on a modification of the machine for the reduction to normal form: \( A_1 \) and \( A_2 \) are convertible if their canonical forms \( B_1 \) and \( B_2 \) have the same outermost constructor and the components of \( B_1 \) and \( B_2 \) are convertible, otherwise not.

The combinators pose one additional problem, however. Consider the combinator \( t \cdot g \cdot \text{App} \). Suppose we have \( \Gamma \vdash t \cdot 1 \rightarrow \Pi(A, B) \) and \( \Gamma \vdash g : A \cdot A' \). To make the combinator \( t \cdot g \cdot \text{App} \) well-formed, we need a type \( B \) such that \( \langle Fst; g; Fst, Snd[A'] \rangle \star B' \leftrightarrow^* B \). But such a \( B' \) cannot be derived from \( \Gamma, t \) and \( g \) in general. This problem occurs because we have omitted the type information in the application but have essentially retained the rather restrictive typing rules for the application when we defined the implicit combinators. The solution is therefore to use the reduction \( g \cdot \text{App} \rightarrow^* \langle \text{Id}, g \cdot \text{Snd} \rangle \cdot \text{App} \), and typecheck the latter combinator, which is the translation of the application in the calculus of constructions. In this case we have \( \langle Fst; g; Fst, Snd[A'] \rangle \star B' \leftrightarrow^* B \), and so we can choose \( B' \) to be \( B \). The combinator \( g \cdot \text{Id} \) causes similar problems. Therefore, we define a map \( \text{st} \), which assigns to every combinator \( e \) the combinator \( e' \) obtained by replacing all sub-combinators \( t \cdot f \star (g \cdot s), \ t \cdot f \star (s_1 \cdot s_2), \ g \cdot \text{App} \), and \( g \cdot \text{Id} \) by \( \text{st}(t; (f; g) \star s), \ \text{st}(t; f \star s_1; f \star s_2), \ \langle \text{Id}, \text{st}(g) \cdot \text{Snd} \rangle \cdot \text{App} \) and \( \text{Id} \) respectively.

So the modified inference system \( \Gamma \vdash_s \), specified in Table 7, will use judgements like \( f \Rightarrow g \), where \( \Rightarrow \) can be either \( \Rightarrow_E \) or \( \Rightarrow_L \), and the above test for convertibility. It specifies an algorithm for type synthesis because it is completely syntax-directed:
Table 7
Type synthesis for combinators for the calculus of constructions

Context morphisms

<table>
<thead>
<tr>
<th>( \Gamma \vdash f: \Gamma' )</th>
<th>( \Gamma', \Gamma \vdash g: \Gamma'' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash f: \Gamma' )</td>
<td>( \Gamma', \Gamma \vdash g: \Gamma'' )</td>
</tr>
<tr>
<td>( \Delta \vdash A )</td>
<td>( \Delta \vdash A )</td>
</tr>
<tr>
<td>( \Gamma, \Gamma' \vdash f: 1 \to A' )</td>
<td>( \Gamma, \Gamma' \vdash st(f \ast A) \leftrightarrow st(A') )</td>
</tr>
<tr>
<td>( \Gamma \vdash f, [A] : A' \to A )</td>
<td>( \Gamma \vdash A \vdash \text{Fst}: \Gamma )</td>
</tr>
</tbody>
</table>

Types

<table>
<thead>
<tr>
<th>( \Gamma \vdash f: \Delta )</th>
<th>( \Delta \vdash A )</th>
<th>( \Gamma \vdash A \vdash B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash f \ast A )</td>
<td>( \Gamma \vdash A \vdash B )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f \ast A )</td>
<td>( \Gamma \vdash A \vdash B )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash f : \Delta )</td>
<td>( \Delta \vdash A )</td>
<td></td>
</tr>
</tbody>
</table>
| \( \Gamma \vdash f : \Delta \) | \( \Delta \vdash A \) | for every combinator \( c \) there is at most one possible derivation \( \Gamma \vdash s c: A, \Gamma \vdash s c \) or \( \Gamma \vdash s c; 1 \to A \) respectively, which is completely determined by the structure of \( c \).

**Theorem 18.** For every context \( \Gamma \) and implicit combinator \( f, A \) and \( t \):

(i) \( \Gamma \vdash s f: A \) implies \( st(\Gamma) \triangleright st(f); st(A) \)

(ii) \( \Gamma \vdash s A \) implies \( st(\Gamma) \triangleright st(A) \)

(iii) \( \Gamma \vdash s t: 1 \to A \) implies \( st(\Gamma) \triangleright st(t): 1 \to st(A) \)

**Proof.** Induction over the definition of \( \vdash s. \) □

The completeness theorem is as expected:

**Theorem 19.** For every implicit combinator \( f, A, t \) and context \( \Gamma \):

(i) \( \Gamma \triangleright f: \Delta \) implies for any well-formed context \( \Gamma' \leftrightarrow * \Gamma \) the existence of a context \( \Delta' \leftrightarrow * \Delta \) such that \( \Gamma' \vdash s f: \Delta' \)

(ii) \( \Gamma \triangleright A \) implies that any well-formed context \( \Gamma' \leftrightarrow * \Gamma \) satisfies \( \Gamma'' \vdash s A \)
(iii) $\Gamma \vdash t : 1 \to A$ implies that for any well-formed context $\Gamma' \leftrightarrow \ast \Gamma$ there exists a type $A' \leftrightarrow \ast A$ such that $\Gamma' \vdash_s t : 1 \to A'$.

**Proof.** Induction over the derivation of $\Gamma \vdash e : A$. □

An abstract machine for type synthesis, which succeeds with result $A$, $A$ or $B$ for a context morphism $f$, a type $A$ and morphism $t$ such that $\Gamma \vdash_s f : A$, $\Gamma \vdash_s A$ and $\Gamma \vdash_s t : 1 \to B$ respectively and returns an error otherwise, can be developed along the same lines as the reduction machines. It has the same architecture except that the context register now contains the machine code of the context $\Gamma$ on the left side of the turnstyle $\vdash_s$. Some transitions can only be activated if the side conditions corresponding to the clauses $st(A) \Rightarrow f \ast B$ or $st(A_1) \leftrightarrow \ast st(A_2)$ are fulfilled. In these causes the appropriate reduction machines can be directly activated as outlined above because they will perform the $st$-function during reduction. The symbol $\leftrightarrow \ast$ denotes the convertibility test based on the reduction machines of the previous chapter.

There are two important aspects of this algorithm for type synthesis. Firstly, it demonstrates the efficiency gains obtained by postponing substitution. Take as an example the combinator $Snd$ corresponding to the variable $x_0$. It has the type $Fst \ast A$, and so if the weakening was not postponed, it would occur at every access to a variable. The machine avoids such unwanted weakening reductions by just keeping the weakening combinator $Fst$ as part of the context and letting the rule for variable access $Fst^k : \langle f, t \rangle \ast Snd \Rightarrow Fst^k \ast t$ push the weakening inside subterms. Secondly, the inference rules for $\Gamma \vdash_s f : A$ explain why in general the type $A$ is included in the context morphism $\langle f, t[A] \rangle$. The reason is that it is in general impossible to derive the type $A$ from a type $B$ convertible to $f \ast A$. In the special case $f = Id$, obviously $A = f \ast A$, and so there is no need to indicate the type $A$.

### 4. Implementation issues

This section describes an implementation of the abstract machines and of the typechecker in ML and compares them with the implementations used in the theorem provers LEGO and Coq, also written in ML. Because the only potentially time-consuming part of the type checking algorithm is the reduction to weak head-normal form, we only examine the efficiency of the reduction machines. The description is rather brief because the main thrust of this work has been to establish the theoretical underpinning of the abstract machines. Much remains to be done at the level of practical implementations. If efficiency is really critical an implementation in a language that is more closely related to the machine architecture like C should be used.

As the translation of a combinator into machine code replaces only some nodes in the graph by a list of nodes and does not change the structure of the graph, the implementation uses an ML-datatype that directly corresponds to the combinators.
The transitions corresponding to the list of nodes are executed whenever the constructor corresponding to the combinator is encountered during reduction. This is easier than the introduction of a separate type of machine instructions and translation functions from them into combinators and vice versa. Efficiency considerations suggest one important difference between the combinators and the ML-datatype, however. The combinators Fst\textsuperscript{k} and Fst\textsuperscript{k} * Snd occur throughout the transition tables for the abstract machines, and hence we introduce special constructors \texttt{fstn} and \texttt{sndn}, which take an integer parameter. The datatype for combinators is actually a sum of four datatypes, one for contexts, context morphisms, types and morphisms. In this way the typechecking algorithm of ML detects any confusion of sorts. The signature for the combinators in the eager case is given in Fig. 1.

The signature for the lazy case replaces the datatype env of \texttt{cm * morphism} by env of \texttt{cm * morphism ref} and adds a \texttt{freeze}-constructor. This change captures sharing: the access to an environment env (\texttt{f}, \texttt{t}) yields \texttt{s} if \texttt{t} is a reference to an ML-expression \texttt{freeze} \texttt{s} and otherwise assigns the value \texttt{freeze} \texttt{u} to \texttt{t} if the code that \texttt{t} references evaluates to \texttt{u}.

A state of the eager machine is represented by a tuple of type

\[
\texttt{cm * comb * NF}
\]

where \texttt{NF} is a product of \texttt{ctype * cmorp131sm} , used as a sum type. In the lazy case the type \texttt{cm ref} replaces the type \texttt{cm} so that an environment can be updated after the evaluation of one of its components. The pattern matching makes it easy to formulate the machine tables in ML: every transition is captured by an alternative in a case-statement.

The theorem prover Coq represents an expression of the calculus of constructions directly as a datatype and uses a call-by-name strategy for their reduction to weak head-normal form or normal form. The operations for variable bindings that capture the notion of contexts are not used for the $\beta$-reduction; instead for an application $\lambda x. A \; t$ the substitution of the argument $t$ for $x$ in $t$ is done as part of the...
signature COMBT =
  sig
    datatype cm = ecm |
      env of cm * morphism | (* <> *)
    fstn of int | (* f, t *)
    comp_cm of cm * cm | (* f;g *)
  and ctype = emptyt | (* 1 *)
    mult_t of cm * ctype | (* f* A *)
    produ of ctype * ctype | (* Pi (A, B) *)
    sum of ctype * ctype | (* Si (A, B) *)
    prop | (* Omega *)
    proof | (* T *)
  and morphism = id_m | (* id *)
    mult_m of cm * morphism | (* f * t *)
    comp_m of morphism * morphism | (* t;s *)
    sndn of int | (* fst*n * snd *)
    cur of ctype * morphism | (* Cur (A, t *)
    app | (* app *)
    forall of ctype*morphism | (* forall (A, t *)
    check of morphism * ctype | (* [A] *)
    pair | (* pair *)
    pi1 | (* pi1 *)
    pi2 | (* pi2 *)
  datatype context = emptyc | pairc of context * ctype
  datatype comb = conv1 of context | conv2 of cm |
    conv3 of ctype | conv4 of morphism
end;

Fig. 1.

two def = \lambda x: Prop. \lambda f: Proof(\alpha \rightarrow \alpha). \lambda x: Proof(\alpha). f(f(x)
one def = \lambda x: Prop. \lambda f: Proof(\alpha \rightarrow \alpha). \lambda x: Proof(\alpha). f
Twelve def = \lambda x: Prop. \lambda f: Proof(\alpha \rightarrow \alpha). \lambda x: Proof(\alpha). 
(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(x))))))))))))))))))))
powern def = twelve nattype (mult two) one
Table 8
Execution times for reduction of test to normal form

<table>
<thead>
<tr>
<th></th>
<th>User</th>
<th>Garbage collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager Machine</td>
<td>4.20s</td>
<td>0.07s</td>
</tr>
<tr>
<td>Lazy Machine</td>
<td>2.85s</td>
<td>4.77s</td>
</tr>
<tr>
<td>LEGO</td>
<td>1102.62s</td>
<td>1118.13s</td>
</tr>
<tr>
<td>Coq</td>
<td>25.65s</td>
<td>66.57s</td>
</tr>
</tbody>
</table>

Note that the normal form of powern is the Church-numeral for 4096, i.e.

\[ \text{bignum} \overset{\text{def}}{=} \lambda x: \text{Prop}. \lambda f: \text{Proof}(x \rightarrow x). \lambda x: \text{Proof}(x). f^{4096} x \]

The normalization command in LEGO and Coq applies only to propositions and their proofs. Hence for measuring the execution times we use a proof of the proposition truep \( \overset{\text{def}}{=} \forall x: \text{Prop}. x \rightarrow x \), namely the term

\[ \text{test} \overset{\text{def}}{=} \text{powern } \text{truep } (\lambda x: \text{Proof}(\text{truep }).x)(\lambda x: \text{Prop}. \lambda p: \text{Proof}(x). p) \]

which has the normal form

\[ \lambda x: \text{Prop}. \lambda x: \text{Proof}(x). x \]

The results are given in Table 8. They are quite encouraging and show that these machines are an efficient alternative to previous ones.

The postponement of weakening is not only necessary for the normalization proof but also improves the efficiency of the machines significantly. As an example, consider the combinator.

\[ \text{largeeomb } \overset{\text{def}}{=} \langle \langle, \llbracket \text{test} \rrbracket \rangle \ast \llbracket \lambda x_256: \text{Prop}. \cdots \lambda x_1: \text{Prop}. x_0 \rrbracket \]

If weakening happens at the beginning of every reduction inside a binding operation in the lazy machine, the reduction of largeeomb requires 256 weakening operations applied to the large combinator \( \llbracket \text{test} \rrbracket \). The execution times for the lazy machine are as follows:

<table>
<thead>
<tr>
<th></th>
<th>User</th>
<th>Garbage collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>with weakening postponed</td>
<td>2.13s</td>
<td>4.97s</td>
</tr>
<tr>
<td>with weakening not postponed</td>
<td>21.17s</td>
<td>29.87s</td>
</tr>
</tbody>
</table>

5. Conclusions

D-categories yield a nice way of representing concepts categorically that are important for the design of abstract machines, like environments or closures. This
yields a uniform way of constructing generalized versions of the CAM and Krivine's machine for the calculus of constructions together with a type checker. Both machines can be easily implemented in a programming language that makes it possible to specify details of the underlying machine as well as in a functional language like ML.

There are at least two directions for further research. First, the standard categorical semantics of linear logic via symmetric monoidal closed categories identifies contexts and tensor products in the same way as the categorical semantics for the typed \( \lambda \)-calculus does. As already mentioned in the introduction, multicategories provide a way of separating these two issues, so it is interesting to see if they can be used as a basis for categorical abstract machines for linear logic. A variant of multicategories suitable for derivation of combinators is described in [8], and the rewriting properties of the combinators are investigated in [20]. Second, we have shown strong normalization only for a restricted reduction relation on the combinators. Although this is sufficient for the construction of abstract machines, it is unsatisfactory from a theoretical viewpoint. There is no obvious reason why explicit substitution should change any proof-theoretic properties of the underlying calculus. One possible way forward might be a categorical understanding of the normalization proof via the glueing construction.

Acknowledgements

I would like to thank my supervisor Andy Pitts for many fruitful discussions and suggestions. P.-L. Curien and Martin Hyland also gave helpful suggestions. I was financially supported by the Studienstiftung des deutschen Volkes and the CLICS-project.

References


