
DEGREE PROJECT, IN MATHEMATICS , SECOND LEVEL
*STOCKHOLM*, *SWEDEN 2015*

# Realizability in Coq

ANDERS LUNDSTEDT

**KTH ROYAL INSTITUTE OF TECHNOLOGY**

**SCI SCHOOL OF ENGINEERING SCIENCES**

# Realizability in Coq

A N D E R S  L U N D S T E D T

Master's Thesis in Mathematics (30 ECTS credits)
Master Programme in Mathematics (120 credits)
Royal Institute of Technology year 2015
Supervisor at Stockholm university: Erik Palmgren
Supervisor at KTH was Bengt Ek
Examiner was Bengt Ek

TRITA-MAT-E 2015:71
ISRN-KTH/MAT/E--15/71--SE

Royal Institute of Technology
*SCI School of Engineering Sciences*

**KTH** SCI
SE-100 44 Stockholm, Sweden

URL: www.kth.se/sci

## Abstract

This thesis describes a Coq formalization of realizability interpretations of arithmetic. The realizability interpretations are based on partial combinatory algebras—to each partial combinatory algebra there is an associated realizability interpretation. I construct two partial combinatory algebras. One of these gives a realizability interpretation equivalent to Kleene's original one, without involving the usual recursion-theoretic machinery.



## Sammanfattning

Den här uppsatsen beskriver en Coq-formalisering av realiserbarhetstolkningar av aritmetik. Realiserbarhetstolkningarna baseras på partiella kombinatoriska algebror—för varje partiell kombinatorisk algebra finns det en motsvarande realiserbarhetstolkning. Jag konstruerar två partiella kombinatoriska algebror. En av dessa ger en realiserbarhetstolkning som är ekvivalent med Kleenes ursprungliga tolkning, men dess konstruktion använder inte det sedvanliga rekursionsteoretiska maskineriet.

# Contents

# 1. Introduction

The goal of this thesis project was to construct in Coq (without additional axioms) a realizability interpretation of arithmetic.

The first realizability interpretation of arithmetic was introduced by Kleene (1945) and Nelson (1947) as a way of formalizing the BHK-interpretation of logic (in the context of arithmetic). Kleene's definition uses natural numbers as codes for partial functions that acts as truth witnesses—"realizers"—of arithmetic statements. This can be generalized by abstracting the role of the realizers. One way of doing this is by using "partial combinatory algebras". Any partial combinatory algebra can be used to define a notion of realizability and for nontrivial partial combinatory algebras this gives non-classical interpretations of arithmetic. Kleene's interpretation is the special case where the partial combinatory algebra is the set of (codes for) partial recursive functions.

My achievement is essentially the Coq formalization (which is included as appendices). The statements and proofs I have formalized are not new. However it was for me a nontrivial task to formulate these such that a Coq formalization was viable. In particular, while free variables in logic are more or less easily handled in non-formalized mathematics, in my formalization they became a major complication. I therefore chose to formalize arithmetic in a first-order logic "without free variables". Furthermore, formalizing Kleene's original interpretation would require a formalization of recursion theory. To avoid this my partial combinatory algebras are "term model" constructions.

The reader who is primarily interested in the formalization itself should probably start with the appendices, where the Coq source is provided and documented. In the main part of the thesis I will give an overview of the mathematics I have formalized. For this purpose, some proofs will be rough outlines, or omitted entirely. For full proofs I refer to the Coq source.

# 2. Heyting arithmetic in a logic "without free variables"[1]

This section describes a formulation of Heyting arithmetic in an intuitionistic first-order logic where the syntax is restricted so that in the rules and axioms the clauses handling free variables are superfluous. It is inspired by and should resemble categorical semantics for first-order logic, see e.g. Jacobs (1999).

It should be relatively straightforward that this formulation is equivalent (with respect to derivability) with the Hilbert system in Troelstra and Schwichtenberg (2000), modulo renaming of variables.

**Definition 1 (Terms).**

For each natural number $n$, the set $\mathrm{Term}^n_{\mathrm{HA}}$ of *terms of arity* $n$ is given inductively:

$$\frac{i < n}{x^n_i \in \mathrm{Term}^n_{\mathrm{HA}}} \qquad \frac{}{0^n \in \mathrm{Term}^n_{\mathrm{HA}}} \qquad \frac{t \in \mathrm{Term}^n_{\mathrm{HA}}}{S(t) \in \mathrm{Term}^n_{\mathrm{HA}}}$$

[1] The formalization corresponding to this section is in Appendix E.

$$\frac{u, v \in \mathrm{Term}^n_{\mathrm{HA}}}{(u + v) \in \mathrm{Term}^n_{\mathrm{HA}}} \qquad \frac{u, v \in \mathrm{Term}^n_{\mathrm{HA}}}{(u \cdot v) \in \mathrm{Term}^n_{\mathrm{HA}}} \ .$$

*Notation.* $t$, $u$, $v$, $w$, $t_0, t_1, \ldots$ will denote arbitrary terms when the arity is unimportant or can be inferred from the context. $t^n$, $u^n$, $v^n$, $w^n$, $t^n_0, t^n_1, \ldots$ will denote arbitrary terms of arity $n$. I will drop parentheses when possible, using the standard order of operations (choice of associativity will not matter).

**Definition 2 (Atomic formulas).**
For each natural number $n$, the set $\mathrm{Atom}^n_{\mathrm{HA}}$ of *atomic formulas of arity n* is given inductively:

$$\frac{}{\bot^n \in \mathrm{Atom}^n_{\mathrm{HA}}} \qquad \frac{u, v \in \mathrm{Term}^n_{\mathrm{HA}}}{u \doteq v \in \mathrm{Atom}^n_{\mathrm{HA}}} \ .$$

Apart from bookkeeping of the set of possibly occurring variables, Definitions 1 and 2 do not differ from usual definitions of terms and atoms in arithmetic. Note the superscripts in $0^n$ and $\bot^n$. I allow myself to drop these when the arity can be inferred from the context.

**Definition 3 (Formulas).**
For each natural number $n$, the set $\mathrm{Form}^n_{\mathrm{HA}}$ of *formulas of arity n* is given inductively:

$$\frac{A \in \mathrm{Atom}^n_{\mathrm{HA}}}{A \in \mathrm{Form}^n_{\mathrm{HA}}} \qquad \frac{A, B \in \mathrm{Form}^n_{\mathrm{HA}}}{(A \wedge B) \in \mathrm{Form}^n_{\mathrm{HA}}} \qquad \frac{A, B \in \mathrm{Form}^n_{\mathrm{HA}}}{(A \vee B) \in \mathrm{Form}^n_{\mathrm{HA}}}$$

$$\frac{A, B \in \mathrm{Form}^n_{\mathrm{HA}}}{(A \to B) \in \mathrm{Form}^n_{\mathrm{HA}}} \qquad \frac{A \in \mathrm{Form}^{n+1}_{\mathrm{HA}}}{\exists A \in \mathrm{Form}^n_{\mathrm{HA}}} \qquad \frac{A \in \mathrm{Form}^{n+1}_{\mathrm{HA}}}{\forall A \in \mathrm{Form}^n_{\mathrm{HA}}} \ .$$

*Notation.* $A$, $B$ and $C$ will denote arbitrary formulas when the arity is unimportant or can be inferred from the context. $A^n$, $B^n$ and $C^n$ will denote arbitrary formulas of arity $n$. I will drop parentheses when possible, using the convention that $\to$ associates to the right and that $\forall$ and $\exists$ bind stronger than $\wedge$ and $\vee$, which bind stronger than $\to$.

The clauses for quantifiers deserve some explanation. The idea is that they implicitly "bind" the variable $x^{n+1}_n$. Write

$$A^{n+1}(x^{n+1}_0, \ldots, x^{n+1}_n)$$

to indicate the $n + 1$ variables that can occur "free" in a formula $A^{n+1}$. Then the formula in standard first-order syntax corresponding to $\exists A^{n+1}$ is intended to be

$$\exists x^{n+1}_n A^{n+1}(x^{n+1}_0, \ldots, x^{n+1}_n),$$

which binds the variable $x^{n+1}_n$.

Definitions 1 to 3 provide a syntax with automatic bookkeeping of the set of variables which can occur "free", viz. the variables that can occur "free" in a formula $A^n$ is $x_0^n, \ldots, x_{n-1}^n$. This convenience comes at a price though, since substitution in quantified formulas cannot be straightforwardly defined, as we will now see.

**Definition 4 (Substitution in terms).**
For a term $t^n$ of arity $n$ and a vector $\mathbf{t} = (t_0^m, \ldots, t_{n-1}^m)$ of $n$ terms of arity $m$, the *substitution of* $\mathbf{t}$ *in* $t^n$, written $t^n \,/\, \mathbf{t}$, is a term of arity $m$ defined by recursion on the structure of $t^n$:

$$\begin{aligned} 0^n \,/\, \mathbf{t} &\coloneqq 0^m, \\ x_i^n \,/\, \mathbf{t} &\coloneqq t_i^m, \\ S(t) \,/\, \mathbf{t} &\coloneqq S(t \,/\, \mathbf{t}), \\ (u + v) \,/\, \mathbf{t} &\coloneqq u \,/\, \mathbf{t} + v \,/\, \mathbf{t}, \\ (u \cdot v) \,/\, \mathbf{t} &\coloneqq u \,/\, \mathbf{t} \cdot v \,/\, \mathbf{t}. \end{aligned}$$

**Definition 5 (Term promotion).**
For a term $t^n$ of arity $n$, the corresponding term of arity $n+1$ is the *promoted term* $\widehat{t^n} \coloneqq t^n \,/\, (x_0^{n+1}, \ldots, x_{n-1}^{n+1})$.

**Definition 6 (Substitution in formulas).**
For a formula $A^n$ of arity $n$ and a vector $\mathbf{t} = (t_0^m, \ldots, t_{n-1}^m)$ of $n$ terms of arity $m$, the *substitution of* $\mathbf{t}$ *in* $A^n$, written $A^n \,/\, \mathbf{t}$, is a formula of arity $m$ defined by recursion on the structure of $A^n$:

$$\begin{aligned} \bot^n \,/\, \mathbf{t} &\coloneqq \bot^m, \\ (u \doteq v) \,/\, \mathbf{t} &\coloneqq u \,/\, \mathbf{t} \doteq v \,/\, \mathbf{t}, \\ (A \wedge B) \,/\, \mathbf{t} &\coloneqq A \,/\, \mathbf{t} \wedge B \,/\, \mathbf{t}, \\ (A \vee B) \,/\, \mathbf{t} &\coloneqq A \,/\, \mathbf{t} \vee B \,/\, \mathbf{t}, \\ (A \to B) \,/\, \mathbf{t} &\coloneqq A \,/\, \mathbf{t} \to B \,/\, \mathbf{t}, \\ (\exists A) \,/\, \mathbf{t} &\coloneqq \exists (A \,/\, (\widehat{t_0^m}, \ldots, \widehat{t_{n-1}^m}, x_m^{m+1})), \\ (\forall A) \,/\, \mathbf{t} &\coloneqq \forall (A \,/\, (\widehat{t_0^m}, \ldots, \widehat{t_{n-1}^m}, x_m^{m+1})). \end{aligned}$$

Again, the clauses for quantifiers deserve some explanation. Write again

$$A^{n+1}(x_0^{n+1}, \ldots, x_n^{n+1})$$

to indicate the variables that can occur "free" in a formula $A^{n+1}$. The intended interpretation of $\exists A^{n+1}$ was

$$\exists x_n^{n+1} A^{n+1}(x_0^{n+1}, \ldots, x_n^{n+1}).$$

We want to substitute $(t_0^m, \ldots, t_{n-1}^m)$ in this formula, obtaining a formula for which

the interpretation is

$$\exists y A^{n+1}(t_0^m, \ldots, t_{n-1}^m, y),$$

where $y$ does not occur in any of $t_0^m, \ldots, t_{n-1}^m$. The obvious choice of $y$ is $x_m^{m+1}$. To get a well-formed formula with this choice we have to promote the terms $t_0^m, \ldots, t_{n-1}^m$. Thus we arrive at

$$(\exists A) \,/\, \mathbf{t} \coloneqq \exists (A \,/\, (\widehat{t_0^m}, \ldots, \widehat{t_{n-1}^m}, x_m^{m+1})).$$

In the following we will only be interested in two special kinds of substitutions: formula promotion and substitution of the last variable.

**Definition 7 (Formula promotion).**
For a formula $A^n$ of arity $n$, the corresponding formula of arity $n+1$ is the *promoted formula* $\widehat{A^n} \coloneqq A^n \,/\, (x_0^{n+1}, \ldots, x_{n-1}^{n+1})$.

**Definition 8 (Substitution of the last variable).**
*Substitution of the last variable for a term* $t^{n+1}$, respectively *for a term* $t^n$, in a formula $A^{n+1}$ is given notation and defined by

$$A^{n+1} \,/\, t^{n+1} \coloneqq A^{n+1} \,/\, (x_0^{n+1}, \ldots, x_{n-1}^{n+1}, t^{n+1})$$

respectively by

$$A^{n+1} \,/\, t^n \coloneqq A^{n+1} \,/\, (x_0^n, \ldots, x_{n-1}^n, t^n).$$

(Note that $A^{n+1} \,/\, t^{n+1}$ has arity $n+1$ and that $A^{n+1} \,/\, t^n$ has arity $n$.) One might wonder why Definition 6 was necessary, when the substitutions we are interested in are given by Definitions 7 and 8. Could we not define substitution of the last variable "more directly", so that reasoning about it becomes less complex? Possibly, but of the approaches I tried this one worked best.

**Definition 9 (Rules).**
For all formulas $A$ and $B$ of same arity and for all formulas $A^{n+1}$, the following rules of inference are valid.

$$\frac{A}{\widehat{A}} \qquad \frac{A \to B \qquad A}{B} \qquad \frac{A^{n+1}}{\forall A^{n+1}} \,.$$

**Definition 10 (Propositional axiom schemas).**
For all formulas $A$, $B$ and $C$ of same arity the following formulas are axioms.

$$\begin{aligned} A \to B \to A, \\ (A \to B \to C) \to (A \to B) \to A \to C, \\ A \to B \to A \wedge B, \\ A \wedge B \to A, \end{aligned}$$

$$
\begin{aligned}
A \wedge B &\to B,\\
A &\to A \vee B,\\
B &\to A \vee B,\\
A \vee B \to (A \to C) \to (B \to C) &\to C,\\
\bot &\to A.
\end{aligned}
$$

**Definition 11 (Quantifier axiom schemas).**
For all formulas $A^{n+1}$ and $B^n$ the following formulas are axioms.

$$
\begin{aligned}
A^{n+1} \,/\, t^n &\to \exists A^{n+1},\\
\forall (A^{n+1} \to \widehat{B^n}) \to \exists A^{n+1} &\to B^n,\\
\forall (\widehat{B^n} \to A^{n+1}) \to B^n &\to \forall A^{n+1},\\
\forall A^{n+1} &\to A^{n+1} \,/\, t^n.
\end{aligned}
$$

Definitions corresponding to Definitions 9 to 11 can be found in Troelstra and Schwichtenberg (2000), except for the "promotion rule", i.e. the one with which one from $A$ infers $\widehat{A}$. This rule is of course superfluous in standard presentations of first-order logic.

Note how we in Definition 11 do not need to consider free variables. E.g. the corresponding standard version of the axiom for existential elimination would be

$$\forall x (A \to B) \to \exists x\, A \to B \quad (x \text{ not free in } B).$$

The intended interpretation of our version of this axiom is

$$
\begin{aligned}
\forall x_n^{n+1} (A(x_0^{n+1}, \ldots, x_n^{n+1}) \to B(x_0^{n+1}, \ldots, x_{n-1}^n))&\\
\to \exists x_n^{n+1}\, A(x_0^{n+1}, \ldots, x_n^{n+1}) \to B(x_0^{n+1}, \ldots, x_{n-1}^n),&
\end{aligned}
$$

so the free variable condition is automatically fulfilled.

The following may not be the "standard" axiomatization of equality. I chose this in favor of a substitution schema, which would be more difficult to reason about due to the complexities of substitution.

**Definition 12 (Equality axioms).**

$$
\begin{aligned}
x_0^1 &\doteq x_0^1,\\
x_0^2 \doteq x_1^2 &\to x_1^2 \doteq x_0^2,\\
x_0^3 \doteq x_1^3 \to x_1^3 \doteq x_2^3 &\to x_0^3 \doteq x_2^3,\\
x_0^2 \doteq x_1^2 &\to S(x_0^2) \doteq S(x_1^2),\\
x_0^4 \doteq x_1^4 \to x_2^4 \doteq x_3^4 &\to x_0^4 + x_2^4 \doteq x_1^4 + x_3^4,\\
x_0^4 \doteq x_1^4 \to x_2^4 \doteq x_3^4 &\to x_0^4 \cdot x_2^4 \doteq x_1^4 \cdot x_3^4.
\end{aligned}
$$

The following are the usual Peano axioms.

**Definition 13 (Peano axioms).**

$$\begin{aligned} S(x_0^1) &\neq 0, \\ S(x_0^2) \doteq S(x_1^2) &\to x_0^2 \doteq x_1^2, \\ x_0^1 + 0 &\doteq x_0^1, \\ x_0^2 + S(x_1^2) &\doteq S(x_0^2 + x_1^2), \\ x_0^1 \cdot 0 &\doteq 0, \\ x_0^2 \cdot S(x_1^2) &\doteq x_0^2 + x_0^2 \cdot x_1^2, \\ A^{n+1} \,/\, 0 \to \forall(A^{n+1} \to A^{n+1} \,/\, S(x_n^{n+1})) &\to \forall A^{n+1}. \end{aligned}$$

We want semantics that allows for non-classical models. The easiest way to achieve this is perhaps by the following (non-standard) definition.

**Definition 14 (Interpretation).**
An *interpretation of Heyting arithmetic* is a set of formulas that does not contain $\bot^0$, that contains all axiom instances and that is closed under rule application.

Defining the standard interpretation is straightforward.

**Definition 15 (Standard term interpretation).**
For a term $t^n$ the *standard term interpretation* is a function $[\![t^n]\!] \colon \mathbb{N}^n \to \mathbb{N}$ defined by recursion on the structure of $t^n$:

$$\begin{aligned} [\![0]\!](\mathbf{x}) &\coloneqq 0, \\ [\![x_i^n]\!](x_0, \ldots, x_{n-1}) &\coloneqq x_i, \\ [\![S(t)]\!](\mathbf{x}) &\coloneqq [\![t]\!](\mathbf{x}) + 1, \\ [\![u + v]\!](\mathbf{x}) &\coloneqq [\![u]\!](\mathbf{x}) + [\![v]\!](\mathbf{x}), \\ [\![u \cdot v]\!](\mathbf{x}) &\coloneqq [\![u]\!](\mathbf{x}) \cdot [\![v]\!](\mathbf{x}). \end{aligned}$$

**Definition 16 (Standard formula interpretation).**
For a formula $A^n$ the *standard formula interpretation* is a relation $[\![A^n]\!]$ on $\mathbb{N}^n$ defined by recursion on the structure of $A^n$:

$$\begin{aligned} [\![\bot]\!] &\coloneqq \emptyset, \\ [\![u \doteq v]\!] &\coloneqq \{\mathbf{x} : [\![u]\!](\mathbf{x}) = [\![v]\!](\mathbf{x})\}, \\ [\![A \wedge B]\!] &\coloneqq [\![A]\!] \cap [\![B]\!], \\ [\![A \vee B]\!] &\coloneqq [\![A]\!] \cup [\![B]\!], \\ [\![A \to B]\!] &\coloneqq \{\mathbf{x} : \mathbf{x} \in [\![A]\!] \implies \mathbf{x} \in [\![B]\!]\}, \\ [\![\exists A]\!] &\coloneqq \{\mathbf{x} : \exists y \in \mathbb{N}, (\mathbf{x}, y) \in [\![A]\!]\}, \\ [\![\forall A]\!] &\coloneqq \{\mathbf{x} : \forall y \in \mathbb{N}, (\mathbf{x}, y) \in [\![A]\!]\}. \end{aligned}$$

Recall that $\mathbb{N}^0$ is (most conveniently defined as) the unit set, so that an interpretation of a formula of arity 0 is either the unit set or the empty set.

**Definition 17 (Standard validity).**
A formula $A^n$ is *valid under the standard interpretation* if $[\![A^n]\!] = \mathbb{N}^n$.

**Theorem 18 (The standard interpretation is an interpretation).**
The set of formulas valid under the standard interpretation is an interpretation.

*Proof.* Straightforward but omitted. □

# 3. Partial combinatory algebras[2]

**Definition 19 (Partial applicative structure).**
A *partial applicative structure*, or *PAS*, is a nonempty set $P$ together with a partial function $\cdot : P \times P \rightharpoonup P$, called application. A PAS is *total* if application is total.

*Notation.* Application will be written with infix notation. I allow myself to identify a PAS with its carrier set. $a$, $b$, $c$, $a_0, a_1, \ldots$ will denote arbitrary elements of a given PAS.

**Definition 20 (Terms over a partial applicative structure).**
For a PAS $P$ and for each natural number $n$, the set $\mathrm{Term}^n_{\mathrm{PAS}}$ of *terms of arity $n$ over $P$* is defined inductively:

$$\frac{i < n}{x^n_i \in \mathrm{Term}^n_{\mathrm{PAS}}} \qquad \frac{a \in P}{a^n \in \mathrm{Term}^n_{\mathrm{PAS}}} \qquad \frac{u \in \mathrm{Term}^n_{\mathrm{PAS}} \qquad v \in \mathrm{Term}^n_{\mathrm{PAS}}}{(uv) \in \mathrm{Term}^n_{\mathrm{PAS}}} \ .$$

In particular, $\mathrm{Term}^0_{\mathrm{PAS}}$ is the set of *closed terms*.

*Notation.* $t$, $u$, $v$, $w$, $t_0, t_1, \ldots$ (possibly primed) will denote arbitrary terms when the arity is unimportant or can be inferred from the context. $t^n$, $u^n$, $v^n$, $w^n$, $t^n_0, t^n_1, \ldots$ (possibly primed) will denote arbitrary terms of arity $n$. I adopt left associativity and drop parentheses when possible, i.e. $uvw \coloneqq (uv)w$. I allow myself to drop superscripts on constants when possible. Any closed term is also naturally a term of any arity, so I will let myself use closed subterms in any term, possibly by adding superscripts to indicate the intended arity—i.e. for a closed term $t$, $t^n$ is the corresponding term of arity $n$.

There is of course the natural notion of "evaluating" closed terms, where the evaluation either gives an element or is undefined. We want a notion of equivalence of closed terms, identifying closed terms evaluating to the same element as well as identifying closed terms whose evaluations are undefined. This is accomplished constructively by the following definition.

[2] The formalization corresponding to this section is in Appendices G and H.

**Definition 21 (Equivalence of closed terms).**
*Equivalence between closed terms* $u$ and $v$ over a PAS $P$, written $u \simeq v$, is the smallest equivalence relation such that for all $c \in P$ and all closed terms $u$ and $v$,

$$uv \simeq c \iff \exists a, b \in P,\, u \simeq a \wedge v \simeq b \wedge a \cdot b = c$$

and

$$u \simeq v \iff \forall a \in P,\, u \simeq a \iff v \simeq a.$$

Constructing the above relation is straightforward: take the reflexive, symmetric, transitive closure of the relation generated by the two clauses. However, from this construction I found it nontrivial to prove the important property

$$a = b \iff a \simeq b.$$

Therefore I chose another construction, for which I refer to the Coq source.

**Definition 22 (Closed term denotation).**
We say that a closed term $t$ *denotes*, written $t\downarrow$, if there is an element $a$ such that $t \simeq a$. A denoting closed term is called a *combinator*.

An important consequence of Definitions 21 and 22 is that application and denotation respects closed term equivalence, i.e.

$$u \simeq u' \implies v \simeq v' \implies uv \simeq u'v'$$

and

$$u \simeq v \implies u\downarrow \implies v\downarrow.$$

This allows us to manipulate equations as one expects to be able to.

**Definition 23 (Substitution in terms).**
For a term $t^n$ and a vector $\mathbf{a} \coloneqq (a_0, \ldots, a_{n-1})$ of $n$ elements, the *substitution of* $\mathbf{a}$ *in* $t^n$, written $t^n \,/\, \mathbf{a}$, is a closed term defined by recursion on the structure of $t^n$:

$$\begin{aligned} a \,/\, \mathbf{a} &\coloneqq a, \\ x_i^n \,/\, \mathbf{a} &\coloneqq a_i, \\ (uv) \,/\, \mathbf{a} &\coloneqq (u \,/\, \mathbf{a})(v \,/\, \mathbf{a}). \end{aligned}$$

**Definition 24 (Partial combinatory algebra).**
A *partial combinatory algebra*, or *PCA*, is a PAS with elements $k$ and $s$ such that for all elements $a$, $b$ and $c$,

$$\begin{aligned} kab &\simeq a, \\ sab &\downarrow, \\ sabc &\simeq ac(bc). \end{aligned}$$

**Example 25 (The trivial PCA).**
*The trivial PCA* has only one element $k = s$. The axiom $sab \downarrow$ becomes $kkk \downarrow$, from which we deduce $k \cdot k = k$, an application which also satisfies the remaining axioms.

One might want to rule out the trivial PCA, by requiring $k$ and $s$ to be distinct. For a version of realizability where the realizability interpretation is "internalized" (see Section 7), this is necessary. For the realizability interpretation in Section 4 it is not. In fact, there the realizability interpretation given by the trivial PCA is equivalent to the standard interpretation (see Example 33).

**Example 26 (Kleene's PCA).**
Given a numbering of the partial recursive functions $\mathbb{N} \to \mathbb{N}$ we may define a PAS as $\mathbb{N}$ with application $n \cdot m$ as the $n$th partial recursive function applied to $m$. If the numbering is admissible[3] then this is a PCA—*Kleene's PCA* $\mathcal{K}$—since with the s-m-n theorem one can show that there are are partial recursive functions filling the roles of $k$ and $s$.[4] (One can show that all admissible numberings give isomorphic PCAs, i.e. PCAs that are isomorphic as PAS's.)

For the remainder of this section we work in an arbitrary PCA.

**Lemma 27 (Identity combinator).**
There is a combinator $\iota$ such that for all closed terms $t$, $\iota t \simeq t$.

*Proof.* Take $\iota \coloneqq skk$. □

**Definition 28 (Term representation).**
For a term $t^n$, its *term representation* is a closed term $[t^n]$ defined by recursion on $n$ and $t$:

$$\begin{aligned} [t^0] &\coloneqq t^0, \\ [a^1] &\coloneqq ka, \\ [x_0^1] &\coloneqq \iota, \end{aligned}$$

[3] Let $\varphi_0, \varphi_1, \ldots$ be Kleene's recursive numbering of the partial recursive functions $\mathbb{N} \to \mathbb{N}$. Another numbering $\psi_0, \psi_1, \ldots$ is *admissible* if there is a recursive bijection $f \colon \mathbb{N} \to \mathbb{N}$ such that $\psi_n = \varphi_{f(n)}$ for all $n$. (There are non-admissible recursive numberings, see e.g. Friedberg, 1958.)

[4] Conversely, if this PAS is a PCA and the numbering is recursive, then the numbering is admissible: a recursive numbering satisfies the s-m-n theorem if and only if it is admissible (Soare, 1999, p. 26) and if a recursive numbering forms a PCA then one can show that its s-m-n theorem holds.

$$
\begin{aligned}
[u^1 v^1] &\coloneqq s[u^1][v^1],\\
[a^{n+2}] &\coloneqq [k^{n+1} a^{n+1}],\\
[x_i^{n+2}] &\coloneqq [k^{n+1} x_i^{n+1}] \quad (i \neq n+1),\\
[x_{n+1}^{n+2}] &\coloneqq [\iota^{n+1}],\\
[u^{n+2} v^{n+2}] &\coloneqq [s^{n+1}([u^{n+2}] x_0^{n+1} \cdots x_n^{n+1})([v^{n+2}] x_0^{n+1} \cdots x_n^{n+1})].
\end{aligned}
$$

**Theorem 29 (Combinatory completeness).**
For all $n$, for all terms $u^{n+1}$ and $v^n$ and for all elements $a_0, \ldots, a_{n-1}$,

$$
\begin{aligned}
&[u^{n+1}] a_0 \cdots a_{n-1} \downarrow,\\
&[v^n] a_0 \cdots a_{n-1} \simeq v^n \,/\, (a_0, \ldots, a_{n-1}).
\end{aligned}
$$

*Proof.* Induction on $n$ and terms. Demonstrating the trickiest case only, i.e. that

$$
[u^{n+2} v^{n+2}] a_0 \cdots a_{n+1} \simeq (u^{n+2} v^{n+2}) \,/\, (a_0, \ldots, a_{n+1}).
$$

We have

$$
\begin{aligned}
&[u^{n+2} v^{n+2}] a_0 \cdots a_{n+1} =\\
&\qquad \text{(by definition)}\\
&[s^{n+1}([u^{n+2}] x_0^{n+1} \cdots x_n^{n+1})([v^{n+2}] x_0^{n+1} \cdots x_n^{n+1})] a_0 \cdots a_{n+1} \simeq\\
&\qquad \text{(induction hypothesis for } n)\\
&\big(\big(s^{n+1}([u^{n+2}] x_0^{n+1} \cdots x_n^{n+1})([v^{n+2}] x_0^{n+1} \cdots x_n^{n+1})\big) \,/\, (a_0, \ldots, a_n)\big)\, a_{n+1} =\\
&\qquad \text{(substitution)}\\
&s([u^{n+2}] a_0 \cdots a_n)([v^{n+2}] a_0 \cdots a_n) a_{n+1} \simeq\\
&\qquad \text{(second } s \text{ axiom)}\\
&[u^{n+2}] a_0 \cdots a_{n+1} ([v^{n+2}] a_0 \cdots a_{n+1}) \simeq\\
&\qquad \text{(induction hypotheses for } u^{n+2} \text{ and } v^{n+2})\\
&u^{n+2} \,/\, (a_0, \ldots, a_{n+1}) v^{n+2} \,/\, (a_0, \ldots, a_{n+1}) =\\
&\qquad \text{(substitution)}\\
&(u^{n+2} v^{n+2}) \,/\, (a_0, \ldots, a_{n+1}).
\end{aligned}
$$

For the remaining cases I refer to the Coq source. □

Combinatory completeness is in fact an equivalent definition of a PCA. To see this, take as $k$ (the denotation of) $[x_0^2]$ and take as $s$ (the denotation of) $[x_0^3 x_2^3 (x_1^3 x_2^3)]$. (Since this direction is easy but the reverse direction is not, the choice of defining formulation is motivated.)

Combinatory completeness basically says that for $n \geq 1$, any term $t^n$ viewed as an $n$-ary function can be "computed" by an element. This gives us strong coding capabilities, so strong that in fact all partial recursive functions are "represented" in any nontrivial

PCA.[5] Or put another way: anything that can be computed can be computed using nothing but distinct $k$ and $s$. This is one of the consequences of the following lemma.

**Lemma 30 (Combinators).**
There are combinators $\pi$ (*pairing*), $\pi_1$ (*left projection*), $\pi_2$ (*right projection*), $\delta$ (*case combinator*), $\sigma$ (*successor combinator*), $\varphi$ (*fixed point combinator*), $\rho$ (*primitive recursion*), $\alpha$ (*addition*), $\mu$ (*multiplication*) and for each natural number $x$, a *natural number combinator* $\overline{x}$, such that for all elements $a$ and $b$ and for all natural numbers $x$ and $y$,

$$\begin{aligned}
\pi_1(\pi ab) &\simeq a,\\
\pi_2(\pi ab) &\simeq b,\\
\delta ab\overline{0} &\simeq a,\\
\delta ab\overline{x+1} &\simeq b,\\
\sigma\overline{x} &\simeq \overline{x+1},\\
\varphi a &\simeq a(\varphi a),\\
\rho ab\overline{0} &\simeq a,\\
\rho ab\overline{x+1} &\simeq b\overline{x}(\rho ab\overline{x}),\\
\alpha\overline{x}\,\overline{y} &\simeq \overline{x+y},\\
\mu\overline{x}\,\overline{y} &\simeq \overline{xy}.
\end{aligned}$$

In addition, in a nontrivial PCA the representation of natural numbers is correct, i.e. in a nontrivial PCA, for all natural numbers $x$ and $y$,

$$\overline{x} \simeq \overline{y} \implies x = y.$$

*Proof.* In some PCAs, e.g. in $\mathcal{K}$, there are natural candidates for some or all of these combinators. Chapter 1 of van Oosten (2008) provides combinators for the general case. I refer to the Coq source for proof that the representation of natural numbers is correct in a nontrivial PCA. □

# 4. Realizability interpretations of Heyting arithmetic[6]

Accounts of realizability on which this section is (loosely) based can be found in Beeson (1985) and van Oosten (2008).

Before I define PCA-based realizability it is instructive to recall Kleene's original definition of realizability. Kleene (1945) defines the relation "$e$ realizes $A$" between natural numbers and (ordinary) first-order formulas by the following seven[7] clauses.

(1) If $A(x_1,\ldots,x_n)$ is a formula containing as free variables exactly $x_1,\ldots,x_n$ in

[5] A function $f\colon \mathbb{N}^n \to \mathbb{N}$ is represented by an element $a$ if for all natural numbers $x_1,\ldots,x_n$ and $y$, $f(x_1,\ldots,x_n) = y$ if and only if $a\overline{x}_1\ldots\overline{x}_n \simeq \overline{y}$.

[6] The formalization corresponding to this section is in Appendix I.

[7] I omit Kleene's redundant clause 5.

order of first free occurrence and if $e$ realizes $\forall x_1, \ldots, x_n, A(x_1, \ldots, x_n)$, then $e$ realizes $A(x_1, \ldots, x_n)$.

(2) An atomic formula without free variables is realized by 0 if it is true.

(3) If $A$ and $B$ are formulas without free variables and if $a$ realizes $A$ and $b$ realizes $B$ then $2^a 3^b$ realizes $A \wedge B$.

(4) If $A$ and $B$ are formulas without free variables and if $a$ realizes $A$ then $2^0 3^a$ realizes $A \vee B$. Similarly, if $b$ realizes $B$ then $2^1 3^b$ realizes $A \vee B$.

(5) If $A$ and $B$ are formulas without free variables and if $e$ is a Gödel number for a partial recursive function $f\colon \mathbb{N} \to \mathbb{N}$ such that for all $a$ realizing $A$, $f(a)$ is defined and realizes $B$, then $e$ realizes $A \to B$.

(6) If $A(x)$ is a formula with a single free variable $x$ and if $a$ realizes $A(n)$ (where $n$ is a constant numeral) then $2^n 3^a$ realizes $\exists x, A(x)$.

(7) If $A(x)$ is a formula with a single free variable $x$ and if $e$ is a Gödel number for a total recursive function $f\colon \mathbb{N} \to \mathbb{N}$ such that for all natural numbers $n$, $f(n)$ realizes $A(n)$, then $e$ realizes $\forall x, A(x)$.

Kleene then defines a realizable formula as a formula $A$ for which there exists a natural number $e$ such that $e$ realizes $A$. A consequence of this definition is that for any formula $A$, not both $A$ and $\neg A$ are realizable. Also, Nelson (1947) shows that all formulas provable in Heyting arithmetic are realizable. Thus in a sense the set of realizable formulas is an interpretation of Heyting arithmetic.

I will generalize (and slightly modify) Kleene's definition, working in an arbitrary PCA and with the formulation of Heyting arithmetic from Section 2.

**Definition 31 (Realizability predicates).**
The *realizability predicate* for a formula $A^n$ is a function $[\![A^n]\!]_r\colon \mathbb{N}^n \to \mathcal{P}(\mathrm{Term}^0_{\mathrm{PAS}})$ defined by recursion on the structure of $A^n$:

$$\begin{aligned}
[\![A]\!]_r(\mathbf{x}) &:= \{t : t\downarrow \wedge\, \mathbf{x} \in [\![A]\!]\} \quad (A \text{ atomic}),\\
[\![A \wedge B]\!]_r(\mathbf{x}) &:= \{t \,:\, \pi_1 t \in [\![A]\!]_r(\mathbf{x})\} \cap \{t \,:\, \pi_2 t \in [\![B]\!]_r(\mathbf{x})\},\\
[\![A \vee B]\!]_r(\mathbf{x}) &:= \{t \,:\, \pi_1 t \simeq \overline{0} \wedge \pi_2 t \in [\![A]\!]_r(\mathbf{x})\} \cup \{t \,:\, \pi_1 t \simeq \overline{1} \wedge \pi_2 t \in [\![B]\!]_r(\mathbf{x})\},\\
[\![A \to B]\!]_r(\mathbf{x}) &:= \{u : u\downarrow \wedge \forall v \in [\![A]\!]_r(\mathbf{x}), uv \in [\![B]\!]_r(\mathbf{x})\},\\
[\![\exists A]\!]_r(\mathbf{x}) &:= \{t : \exists y \in \mathbb{N}, \pi_1 t \simeq \overline{y} \wedge \pi_2 t \in [\![A]\!]_r(\mathbf{x}, y)\},\\
[\![\forall A]\!]_r(\mathbf{x}) &:= \{t : \forall y \in \mathbb{N}, t\overline{y} \in [\![A]\!]_r(\mathbf{x}, y)\}.
\end{aligned}$$

**Definition 32 (Realizable formulas).**
A closed term $t$ is a *realizer* for a formula $A^n$, written $t\ \mathbf{r}\ A^n$, if for all natural numbers $x_1, \ldots, x_n$,

$$t\overline{x_1} \cdots \overline{x_n} \in [\![A^n]\!]_r(x_1, \ldots, x_n).$$

A formula $A^n$ is *realizable* if it has a realizer.

**Example 33 (Realizability with the trivial PCA).**
The realizable formulas for the trivial PCA are precisely the formulas valid under the standard interpretation. (Proved by induction on formulas, with any nontrivial details left to the reader.)

**Example 34 (Realizability with $\mathcal{K}$).**
The set of realizable formulas for Kleene's $\mathcal{K}$ is the same as the set of realizable formulas in Kleene (1945).

Example 34 is a nontrivial result, since my definition does not directly translate to Kleene's. In particular Kleene's definition has a natural number as its own representative, while Definition 31 uses the representatives given by the general proof of Lemma 30. One proves (by induction on formulas) the result by showing that realizers can be translated between the two definitions.

In general, the set of realizable formulas will depend on the PCA. However it will always be an interpretation (in the sense of Definition 14), as I will now show.

**Definition 35 (Realizability term interpretation).**
For each HA term $t^n$, its *realizability term interpretation* is a combinator $[\![t^n]\!]_r$ defined by recursion on the structure of $t^n$:

$$\begin{aligned}
[\![0]\!]_r &\coloneqq [\overline{0}^n], \\
[\![x_i^n]\!]_r &\coloneqq [x_i^n], \\
[\![S(t)]\!]_r &\coloneqq [\sigma([\![t]\!]_r x_0^n \cdots x_{n-1}^n)], \\
[\![u + v]\!]_r &\coloneqq [\alpha([\![u]\!]_r x_0^n \cdots x_{n-1}^n)([\![v]\!]_r x_0^n \cdots x_{n-1}^n)], \\
[\![u \cdot v]\!]_r &\coloneqq [\mu([\![u]\!]_r x_0^n \cdots x_{n-1}^n)([\![v]\!]_r x_0^n \cdots x_{n-1}^n)].
\end{aligned}$$

**Lemma 36 (Soundness of realizability term interpretation).**
For all HA terms $t^n$ and all natural numbers $x_1, \ldots, x_n$,

$$[\![t^n]\!]_r \overline{x_1} \cdots \overline{x_n} \simeq \overline{[\![t^n]\!](x_1, \ldots, x_n)}.$$

*Proof.* Use properties of $\sigma$, $\alpha$ and $\mu$ (Lemma 30) and induction on the structure of $t^n$. □

**Lemma 37 (Rules realizable).**
The set of realizable formulas is closed under application of the rules in Definition 9.

*Proof.*

$$\frac{t \mathbf{r} A^n}{[t x_0^{n+1} \cdots x_{n-1}^{n+1}] \mathbf{r} \widehat{A^n}}$$

$$\frac{u \mathbf{r} A^n \to B^n \qquad v \mathbf{r} A^n}{[u x_0^n \cdots x_{n-1}^n (v x_0^n \cdots x_{n-1}^n)] \mathbf{r} B}$$

$$\frac{t \mathbf{r} A^{n+1}}{t \mathbf{r} \forall A^{n+1}}.$$

□

**Lemma 38 (Propositional axioms realizable).**
All propositional axioms in Definition 10 are realizable.

*Proof.*

$$\begin{aligned}
[k^n] \mathbf{r}\ & A^n \to B^n \to A^n,\\
[s^n] \mathbf{r}\ & (A^n \to B^n \to C^n) \to (A^n \to B^n) \to A^n \to C^n,\\
[\pi^n] \mathbf{r}\ & A^n \to B^n \to A^n \wedge B^n,\\
[\pi_1^n] \mathbf{r}\ & A^n \wedge B^n \to A^n,\\
[\pi_2^n] \mathbf{r}\ & A^n \wedge B^n \to B^n,\\
[\pi^n \overline{0}] \mathbf{r}\ & A^n \to A^n \vee B^n,\\
[\pi^n \overline{1}] \mathbf{r}\ & B^n \to A^n \vee B^n,\\
[[\delta x_1^3 x_2^3 (\pi_1 x_0^3)(\pi_2 x_0^3)]^n] \mathbf{r}\ & A^n \vee B^n \to (A^n \to C^n) \to (B^n \to C^n) \to C^n,\\
[k^n] \mathbf{r}\ & \bot \to A^n.
\end{aligned}$$

□

**Lemma 39 (Quantifier axioms realizable).**
All quantifier axioms in Definition 11 are realizable.

*Proof.*

$$\begin{aligned}
[\pi([\![t^n]\!]_r x_0^{n+1} \cdots x_{n-1}^{n+1}) x_n^{n+1}] \mathbf{r}\ & A^{n+1} \,/\, t^n \to \exists A^{n+1},\\
[[x_0^2 (\pi_1 x_1^2)(\pi_2 x_1^2)]^n] \mathbf{r}\ & \forall (A^{n+1} \to \widehat{B^n}) \to \exists A^{n+1} \to B^n,\\
[[x_0^3 x_2^3 x_1^3]^n] \mathbf{r}\ & \forall (\widehat{B^n} \to A^{n+1}) \to B^n \to \forall A^{n+1},\\
[x_n^{n+1} ([\![t^n]\!]_r x_0^{n+1} \cdots x_{n-1}^{n+1})] \mathbf{r}\ & \forall A^{n+1} \to A^{n+1} \,/\, t^n.
\end{aligned}$$

□

**Lemma 40 (Self realizable formulas).**
Implication chains $A_1 \to \cdots \to A_{n+1}$ of atomic formulas are realizable if they are valid in the standard interpretation.

*Proof.* If $A_1 \to \cdots \to A_{n+1}$ is valid in the standard interpretation then

$$[k^{n+m}] \mathbf{r}\ A_1^m \to \ldots \to A_{n+1}^m.$$

□

**Lemma 41 (Equality axioms realizable).**
All equality axioms in Definition 12 are realizable.

*Proof.* Lemma 40. □

**Lemma 42 (Peano axioms realizable).**
All Peano axioms in Definition 13 are realizable.

*Proof.* All axioms except induction are handled by Lemma 40. Induction is realized by $[\rho^n]$. □

**Theorem 43 (Realizability is an interpretation).**
The set of realizable formulas is an interpretation.

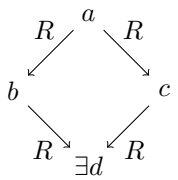
*Proof.* Lemmas 38, 39, 41 and 42 and the trivial fact that $\bot^0$ is not realizable. □

# 5. Rewriting with K and S[8]

This section presents a rewriting system with terms built from constants $K$ and $S$—more or less combinatory logic without variables. I will define two rewriting relations, or "reduction relations". The first will be "non-deterministic", in the sense that a term may be reduced in more than one way. It will also be "lazy", in the sense that a reduction may discard subterms, thus effectively skipping reductions in these. The second rewriting relation will be "deterministic"—a term can be reduced to at most one other term—and "eager"—if possible reductions must be made in subterms first.

With non-deterministic reduction we will be able to construct a total PCA on the set of terms modulo equivalence under non-deterministic reduction. With eager deterministic reduction we will be able to construct a (non-total) PCA on the set of normal terms (the set of terms that cannot be further reduced).[9]

**Definition 44 (The diamond property).**




A binary relation $R$ has the *diamond property*, if $aRb$ and $aRc$ implies the existence of a $d$ such that $bRd$ and $cRd$.

**Definition 45 (Confluence).**

A binary relation $R$ is *confluent* if its transitive reflexive closure $R^*$ has the diamond property.

**Lemma 46 (The diamond property implies confluence).**

For all binary relations $R$, the diamond property of $R$ implies the diamond property of its transitive reflexive closure $R^*$.

*Proof.* See Pollack (1995). □

**Definition 47 (KS-terms).**

The set $\mathrm{Term}_{\mathrm{KS}}$ of *KS-terms* is defined inductively:

$$T := K \mid S \mid (TT).$$

*Notation.* $T, U, V, W, X, Y, Z, T_1, T_2, \ldots$ (possibly primed) will denote arbitrary terms. I adopt left associativity and drop parentheses when possible.

[8] The formalizations corresponding to this section is in Appendices J and K.

[9] The distinction "eager/lazy" is what matters when constructing these PCAs, not the distinction "deterministic/non-deterministic". One can construct the same total PCA using a "lazy deterministic" reduction and the same non-total PCA using an "eager non-deterministic" reduction. However the corresponding constructions turn out to be more complicated.

**Definition 48 (Non-deterministic reduction).**
The *one step reduction relation* $>$ between terms is defined inductively:

$$\frac{}{KUV > U} >_K \quad \frac{}{SUVW > UW(VW)} >_S$$

$$\frac{U > U'}{UV > U'V} >_{\text{left}} \quad \frac{V > V'}{UV > UV'} >_{\text{right}}.$$

The transitive reflexive closure $>^*$ is the *reduction relation*.

**Example 49 (Non-determinism).**
"Non-determinism" is a consequence of $>_{\text{left}}$ and $>_{\text{right}}$. If $U > U'$ and $V > V'$ then we have both $UV > U'V$ (by $>_{\text{left}}$) and $UV > UV'$ (by $>_{\text{right}}$). For example, we have $KSK >_K S$ so by $>_{\text{left}}$ we have

$$KSK(KSK) > S(KSK)$$

but by $>_{\text{right}}$ we then also have

$$KSK(KSK) > KSKS.$$

**Example 50 (Laziness).**
"Laziness" is a consequence of $>_K$. If $V > V'$ then $KUV >_{\text{right}} KUV'$ but this reduction step may be "skipped" since $KUV >_K U$. For example by $>_{\text{right}}$ we have

$$KS(KSK) > KSS$$

but by $>_K$ we also have

$$KS(KSK) > S.$$

Observe that in the above examples, even if non-determinism gives $U > V$ and $U > V'$ (or more generally: $U >^* V$ and $U >^* V'$) for distinct $V$ and $V'$ it is possible to continue these reductions, reaching a common term $W$. For example, continuing the reductions in Example 49 we have

$$KSK(KSK) > S(KSK) > SS$$

and also

$$KSK(KSK) > KSKS > SS.$$

The following theorem states that this holds generally. The history of the method used in the proof is somewhat hard to trace. It first appeared in Martin-Löf's unpublished "A theory of types" (Martin-Löf, 1971), where he describes the proof as based on a technique shown to him by Tait.

**Theorem 51 (Confluence of the one step reduction relation).**
The one step reduction relation is confluent, i.e. for all terms $U$, $V$ and $V'$, if $U >^* V$ and $U >^* V'$ then there exists a term $W$ such that $V >^* W$ and $V' >^* W$.

*Proof.* We cannot apply Lemma 46 directly, since the diamond property does not hold for one step reduction. However if we can find a relation with the diamond property and the same transitive reflexive closure as one step reduction then we are done. To this end, define *one step reflexive parallel reduction* inductively:

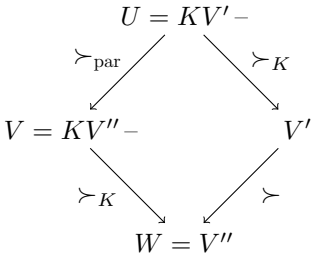
$$\frac{}{KUV \succ U}\,{\scriptstyle \succ_K} \qquad \frac{}{SUVW \succ UW(VW)}\,{\scriptstyle \succ_S}$$

$$\frac{}{U \succ U}\,{\scriptstyle \succ_{\text{refl}}} \qquad \frac{U \succ U' \qquad V \succ V'}{UV \succ U'V'}\,{\scriptstyle \succ_{\text{par}}}\ .$$

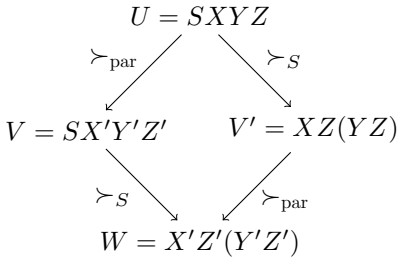
It is quite straightforward that the transitive reflexive closure of this relation is the reduction relation—see Pollack (1995) or the Coq source. Thus it remains to show the diamond property, i.e. that if $U \succ V$ and $U \succ V'$ then there is a $W$ such that $V \succ W$ and $V' \succ W$. We show this by induction on $U \succ V$ followed by case analysis on $U \succ V'$ (omitting redundant cases).

Case $U \succ_{\text{refl}} V$: Then $U = V$. Take $W = V'$, then $V \succ W$ by hypothesis and $V' \succ W$ by $\succ_{\text{refl}}$.

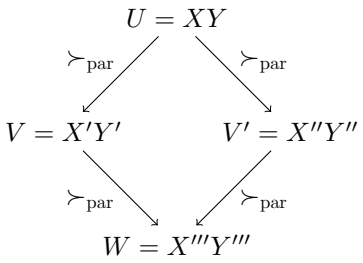
Case $U \succ_{\text{par}} V$, $U \succ_K V'$: Then we have the following situation.




Case $U \succ_{\text{par}} V$, $U \succ_S V'$: Then we have the following situation, where $X \succ X'$, $Y \succ Y'$ and $Z \succ Z'$.



Case $U \succ_{\text{par}} V$, $u \succ_{\text{par}} v'$ : Then we have the following situation, where $X' \succ X'''$, $X'' \succ X'''$, $Y' \succ Y'''$ and $Y'' \succ Y'''$ by induction hypothesis.




Case $U \succ_K V$, $U \succ_S V'$: Then we have the contradiction $U = KV{-}{=}S{-}{-}{-}$ (each dash a distinct variable) so this case is not possible.

Cases $U \succ_K V$, $U \succ_K V'$ and $U \succ_S V, U \succ_S V'$: Then necessarily $V = V'$, so take $W = V$. Then $V \succ W$ and $V' \succ W$ by $\succ_{\text{refl}}$. □

**Definition 52 (Convertibility).**
Two terms $U$ and $V$ are *convertible*, written $U \sim V$, if there is a term $W$ such that $U >^* W$ and $V >^* W$.

**Theorem 53 (Properties of convertibility).**
Convertibility is an equivalence relation that respects concatenation of terms, i.e.

$$U \sim U' \implies V \sim V' \implies UV \sim U'V'.$$

*Proof.* Reflexivity and symmetry are immediate. Transitivity follows from confluence (Theorem 51). By induction on $U >^* U'$ one shows

$$U >^* U' \implies UV >^* U'V.$$

Similarly by induction on $V >^* V'$ one shows

$$V >^* V' \implies UV >^* UV'.$$

It then follows that

$$U >^* U' \implies V >^* V' \implies UV >^* U'V',$$

from which it follows that concatenation respects convertibility. □

**Definition 54 (Deterministic eager reduction).**
The *eager reduct* $\text{red}(T)$ of a term $T$ is either a term or $\bot$, defined by

$$\begin{aligned} \text{red}(UV) &= \text{red}(U)V && \text{if } \text{red}(U) \neq \bot, \\ \text{red}(UV) &= U\,\text{red}(V) && \text{if } \text{red}(U) = \bot \text{ and } \text{red}(V) \neq \bot, \\ \text{red}(KUV) &= U && \text{if } \text{red}(U) = \text{red}(V) = \bot, \end{aligned}$$

$$\begin{aligned} \mathrm{red}(SUVW) &= UW(VW) && \text{if } \mathrm{red}(U) = \mathrm{red}(V) = \mathrm{red}(W) = \bot, \\ \mathrm{red}(T) &= \bot && \text{otherwise.} \end{aligned}$$

A term $U$ *eagerly one step reduces* to a term $V$, written $U \gg V$, if $\mathrm{red}(U) = V$. The transitive reflexive closure $\gg^*$ of this relation is the *eager reduction relation.*

"Determinism" of eager reduction is a trivial consequence of Definition 54, since eager reduction so defined is functional. "Eagerness" is the fact that $KUV \gg U$ and $SUVW \gg UW(VW)$ hold if and only if subterms cannot be reduced, i.e.

$$KUV \gg U \iff \mathrm{red}(U) = \mathrm{red}(V) = \bot$$

and

$$SUVW \gg UW(VW) \iff \mathrm{red}(U) = \mathrm{red}(V) = \mathrm{red}(W) = \bot.$$

**Theorem 55 (Confluence of the eager one step reduction relation).**
The eager reduction relation is confluent, i.e. for all terms $U$, $V$ and $V'$, if $U \gg^* V$ and $U \gg^* V'$ then there exists a term $W$ such that $V \gg^* W$ and $V' \gg^* W$.

*Proof.* Any functional relation is trivially confluent. □

**Lemma 56 (Normal term conditions).**
For a term $U$, the following are equivalent.

(1) There is no $V$ such that $U > V$.

(2) There is no $V$ such that $U \gg V$.

(3) $\mathrm{red}(U) = \bot$.

*Proof.* Omitted. □

**Definition 57 (Normal terms).**
A term for which the conditions in Lemma 56 hold is a *normal term.* The set of normal terms is denoted $\mathcal{NT}$. If $U >^* V$ and $V$ is normal then $V$ is a *normal form* of $U$. If $U \gg^* V$ and $V$ is normal then $V$ is a *strict normal form* of $U$.

**Theorem 58 (Uniqueness of normal forms).**
Each term has at most one (strict) normal form.

*Proof.* Suppose $T >^* U$ and $T >^* V$ with $U$ and $V$ normal. By Theorem 51, there is $W$ such that $U >^* W$ and $V >^* W$, i.e. such that $U = T_0 > \cdots > T_n = W$ and $V = T_0 > \cdots > T_m = W$ for some $n, m \geq 0$. Since $U$ and $V$ are normal, we must have $n = m = 0$, i.e. $U = V = W$. Uniqueness of strict normal forms follow in the same way from Theorem 55. □

**Example 59 (A term with no normal form).**
Let $I \coloneqq SKK$. Then $IT >^* T$ for all terms $T$. $I$ is normal and thus so is $SII$. Thus when reducing $SII(SII)$ we will always end up in the cycle

$$SII(SII) > I(SII)(I(SII)) >^* SII(SII).$$

Thus $SII$ has no normal form.

It is trivial that eager reduction implies lazy reduction, i.e. if $U \gg V$ then $U > V$. Thus any strict normal form is also a normal form. Thus Example 59 also shows that $SII$ has no strict normal form. However, as the following example shows, normal forms need not be strict normal forms.

**Example 60 (A term with a normal form but no strict normal form).**
Consider the term $SK(SII)(SII)$, where $I$ is as in Example 59. We have

$$SK(SII)(SII) \gg K(SII)(SII(SII)).$$

Since $SII(SII)$ has no normal form, a continued eager reduction will go on forever. Thus $SK(SII)(SII)$ has no strict normal form. On the other hand,

$$SK(SII)(SII) > K(SII)(SII(SII)) >_K SII$$

which is normal. Thus $SK(SII)(SII)$ has a normal form. (Notice how the "laziness" of $>_K$ is the key here.)

**Lemma 61 (Induction principle for normal terms).**
If $T$ is normal then either

(1) $T = K$, or

(2) $T = S$, or

(3) $T = KU$ with $U$ normal, or

(4) $T = SU$ with $U$ normal, or

(5) $T = SUV$ with $U$ and $V$ normal.

Consequently we have the following induction principle for propositional functions $P$ on terms:

$$\begin{aligned}
&P(K) \implies P(S) \implies \\
&(\forall T \in \mathcal{NT}, P(T) \implies P(KT)) \implies \\
&(\forall T \in \mathcal{NT}, P(T) \implies P(ST)) \implies \\
&(\forall U, V \in \mathcal{NT}, P(U) \implies P(V) \implies P(SUV)) \implies \\
&\forall T \in \mathcal{NT}, P(T).
\end{aligned}$$

*Proof.* Omitted. □

There is a natural way in which any KS-term is a closed PAS-term over any PCA.

**Definition 62 (Interpretation of KS-terms as closed PCA-terms).**
The *interpretation* of a KS-term $T$ as a PAS-term $\iota(T)$ over any PCA is defined by recursion on $T$:

$$\begin{aligned}\iota(K) &\coloneqq k,\\ \iota(S) &\coloneqq s,\\ \iota(UV) &\coloneqq \iota(U)\iota(V).\end{aligned}$$

**Theorem 63 (Strict reduction and closed term equivalence).**
If $U \gg^* V$ then $\iota(U) \simeq \iota(V)$.

*Proof.* Omitted. □

# 6. Two partial combinatory algebras[10]

**Lemma 64 (The KS-term PAS).**
The set of KS-terms is a total nontrivial PAS $\mathcal{T}$, with equality interpreted by convertibility and application defined as $U \cdot V = UV$.

*Proof.* By Theorem 53 this is a well-defined PAS. Since $K \not\sim S$, it is nontrivial. Totality is immediate. □

**Theorem 65 (The KS-term PCA).**
$\mathcal{T}$ is a PCA, with $K$ and $S$ filling the roles of $k$ and $s$.

*Proof.* Since $\mathcal{T}$ is total, the second axiom is immediate and the first and third axioms become $KUV \sim U$ and $SUVW \sim UW(VW)$, respectively, which are trivial. □

**Lemma 66 (The normal KS-term PAS).**
$\mathcal{NT}$ is a nontrivial PAS, with application $U \cdot V = W$ defined as $UV \gg^* W$.

*Proof.* By Theorem 58, application so defined is functional. Since $K$ and $S$ are distinct normal terms the PAS is nontrivial. □

To prove that $\mathcal{NT}$ is a PCA, it convenient to start with its instance of Theorem 63 (which we otherwise obtain *after* we have proved that $\mathcal{NT}$ is a PCA).

**Definition 67 (Interpretation of KS-terms as closed PAS terms).**
The *interpretation* of a KS-term $T$ as a PAS-term $\iota(T)$ over $\mathcal{NT}$ is defined by recursion on $T$:

$$\begin{aligned}\iota(K) &\coloneqq K,\\ \iota(S) &\coloneqq S,\\ \iota(UV) &\coloneqq \iota(U)\iota(V).\end{aligned}$$

[10] The formalizations corresponding to this section is in Appendices L and M.

**Lemma 68 (Equivalence of closed terms over $\mathcal{NT}$).**
If $U \gg^* V$ then $\iota(U) \simeq \iota(V)$.

*Proof.* Omitted. □

**Theorem 69 (The normal KS-term PCA).**
$\mathcal{NT}$ is a PCA, with $K$ and $S$ filling the roles of $k$ and $s$.

*Proof.* By Lemma 68, it suffices to show that $K$ and $S$ are normal; that $KUV \gg^* U$ for normal $U$ and $V$; that $SUV$ is normal for normal $U$ and $V$; and that $SUVW \gg^* UW(VW)$ for normal $U$, $V$ and $W$. These are trivial. □

*Remark.* If in Lemma 66 we would define application $U \cdot V = W$ as $UV >^* W$, we would still have a PAS. However we would not have a PCA, in a sense because the analogue of Lemma 68 would fail. In particular, since $SKUV > KV(UV) > V$ and since $SKU$ is normal for normal $U$, with application so defined we would have $SKUV \simeq V$. However as Example 59 shows there are normal $U$ and $V$ such that $UV$ has no normal form and for such $U$ and $V$ we would not have $KV(UV) \simeq V$, since $UV$ would not denote. Thus for such $U$ and $V$ we would not have $SKUV \simeq KV(UV)$. Thus the axiom $sabc \simeq ac(bc)$ would not hold. (Cf. Example 60.)

# 7. Comparison with Kleene's original realizability interpretation[11]

Nelson (1947) shows that Kleene's realizability interpretation can be "internalized". This means that for each formula $A$ of Heyting arithmetic we can construct a formula $A^r$ such that Heyting arithmetic proves $A^r$ if and only if $A$ is realizable. The realizability interpretation in Section 4 is not of this kind, however it is possible to achieve similar results. Beeson (1985) shows how to define such "internal" realizability notions for an extension of a first-order intuionistic theory for PCAs. I suspect that formalizing this would require considerable effort, since one either has to prove many results directly in the object theory, or prove sufficiently strong completeness theorems.

From Section 6 we have two different PCAs $\mathcal{T}$ and $\mathcal{NT}$, thus (by Theorem 43) we also have two realizability interpretations of arithmetic. How do these two interpretations and Kleene's interpretation compare to each other (with respect to the set of true formulas)? Since $\mathcal{T}$ is total it is not isomorphic to $\mathcal{K}$. However I do not know whether non-isomorphic PCAs necessarily give different realizability interpretations, but at least I can show that $\mathcal{NT}$ and $\mathcal{K}$ give equivalent realizability interpretations.

**Theorem 70 ($\mathcal{NT}$ and $\mathcal{K}$ are isomorphic).**
$\mathcal{NT}$ and $\mathcal{K}$ are isomorphic as PAS's.

*Proof.* Choose a primitive recursive bijection $i\colon \mathcal{NT} \to \mathbb{N}$ with a primitive recursive inverse. By Lemma 61 we may define application on $\mathbb{N}$ by the following clauses:

$$i(K) \cdot x \coloneqq i(Ki^{-1}(x)),$$

[11] The results in this section have not been formalized. I claim no originality but I have not found these (albeit somewhat trivial) results stated or proved elsewhere.

$$
\begin{aligned}
i(S) \cdot x &\coloneqq i(Si^{-1}(x)),\\
i(KT) \cdot x &\coloneqq i(T),\\
i(ST) \cdot x &\coloneqq i(STi^{-1}(x)),
\end{aligned}
$$

and with $i(SUV) \cdot x$ given in pseudocode:

$a \leftarrow i(U) \cdot x$
$b \leftarrow i(V) \cdot x$
$c \leftarrow a \cdot b$
**return** $c$

(Note that this algorithm is not guaranteed to terminate, in which case $i(SUV) \cdot x$ is undefined.)

By construction $\mathbb{N}$ with application so defined is a PCA isomorphic to $\mathcal{NT}$. Furthermore application is partial recursive, i.e. $x \mapsto n \cdot x$ is partial recursive for all $n$. Thus to show isomorphism with $\mathcal{K}$ it suffices to show that for each partial recursive $f\colon \mathbb{N} \to \mathbb{N}$ there is a natural number $n$ such that for all natural numbers $x$ and $y$ we have $n \cdot x = y$ if and only if $f(x) = y$.

Since $i$ and $i^{-1}$ are primitive recursive, using the power of combinatory completeness (Theorem 29) and combinators (Lemma 30) we may find natural numbers $e$ and $e^{-1}$ such that

$$ex \simeq \overline{x}$$

and

$$e^{-1}\overline{x} \simeq x$$

for all natural numbers $x$. Furthermore we can arrange such that $e^{-1}x$ does not denote if $x$ is not a natural number representation.

Suppose $f\colon \mathbb{N} \to \mathbb{N}$ is partial recursive. Then $f$ is numerically representable, i.e. there is a natural number $\underline{f}$ such that for all natural numbers $x$ and $y$ we have $f(x) = y$ if and only if $\underline{f} \cdot \overline{x} = \overline{y}$. Then

$$[e^{-1}(\underline{f}(ex_0^1))]x \simeq e^{-1}(\underline{f}(ex)) \simeq e^{-1}(\underline{f}\overline{x}).$$

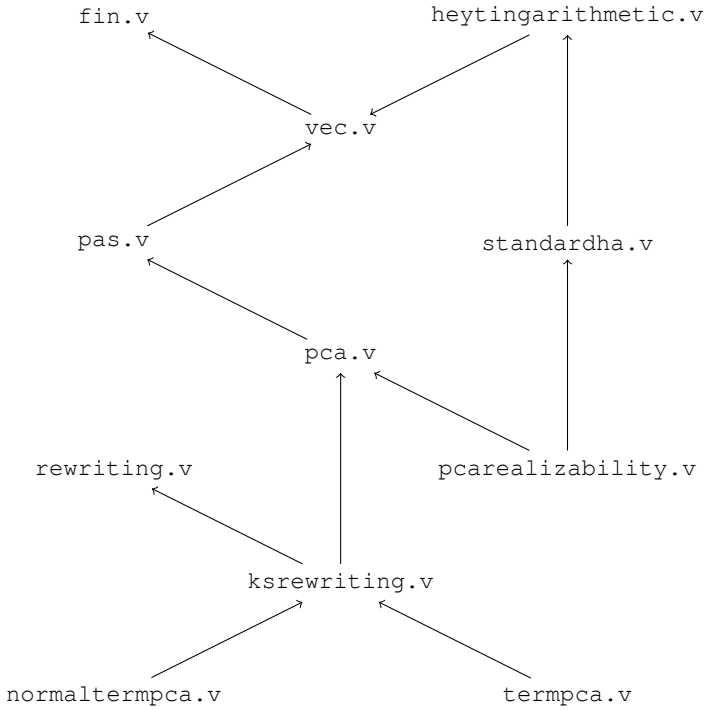
Let $n$ be the denotation of $[e^{-1}(\underline{f}(ex_0^1))]$. Then for all $x$ and $y$ we have $f(x) = y$ if and only if $nx \simeq e^{-1}\overline{y}$, i.e. if and only if $n \cdot x = y$. □

As an immediate corollary we have the following.

**Theorem 71 (The $\mathcal{NT}$ and $\mathcal{K}$ realizability interpretations are equivalent).** The sets of realizable formulas in the $\mathcal{NT}$ and $\mathcal{K}$ realizability interpretations are equal.

# A. How to obtain and typecheck the formalization

I have licensed the Coq source under the MIT license (Appendix B). The formalization is split into eleven Coq modules with the following dependency graph.




**fin.v**
: My own implementation of finite types.

**vec.v**
: My own vector implementation.

**heytingarithmetic.v**
: Syntax and semantics of Heyting arithmetic (corresponding to part of Section 2).

**standardha.v**
: Standard interpretation of Heyting arithmetic (corresponding to part of Section 2).

**pas.v**
: Partial applicative structures (corresponding to part of Section 3).

**pca.v**
: Partial combinatory algebras (corresponding to part of Section 3).

**pcarealizability.v**
: PCA-based realizability interpretations of Heyting arithmetic (corresponding to Section 4).

**rewriting.v**
: Some results relating to rewriting systems (corresponding to part of Section 5), in particular a formalization of Lemma 46.

**ksrewriting.v**
: Rewriting with $K$ and $S$ (corresponding to part of Section 5).

**termpca.v**
: A construction of a total PCA based on the set of KS-terms (corresponding to part of Section 6).

**normaltermpca.v**
: A construction of a PCA based on the set of normal KS-terms (corresponding to part of Section 6).

One way to obtain the source is from my GitHub repository.[12] Alternatively I can provide the source on request. A third option is to copy the source from the appendices that follow.

To typecheck the formalization one compiles the modules. A makefile is in the GitHub repository. Alternatively one may use the utility `coq_makefile` to create one:

```
coq_makefile -o Makefile \
    fin.v \
    heytingarithmetic.v \
    ksrewriting.v \
    normaltermpca.v \
    pas.v \
    pcarealizability.v \
    pca.v \
    rewriting.v \
    standardha.v \
    termpca.v \
    vec.v
```

A third option is to simply compile the modules in an order that compiles any module's dependencies before the module itself, e.g.:

```
coqc \
    fin.v \
    vec.v \
    heytingarithmetic.v \
    standardha.v \
    pas.v \
    pca.v \
    pcarealizability.v \
```

[12] `https://github.com/anderslundstedt/pca-realizability`. The commit corresponding to this thesis is commit `6c52ac692e50605a41d7a728bf9b595015e82fbd` on the master branch.

```
    rewriting.v \
    ksrewriting.v \
    normaltermpca.v \
    termpca.v
```

The modules have been verified to compile with Coq 8.4pl6. Once compiled one should be able to step through the modules in any interactive Coq environment.

To achieve somewhat nice notation I used several non-ASCII characters in the source:[13]

| Character | Unicode code point | Character | Unicode code point |
|---|---|---|---|
| § | U+00A7 | → | U+2192 |
| ¬ | U+00AC | ∀ | U+2200 |
| α | U+03B1 | ∃ | U+2203 |
| δ | U+03B4 | ∘ | U+2218 |
| ι | U+03B9 | ∧ | U+2227 |
| κ | U+03BA | ∨ | U+2228 |
| λ | U+03BB | ≃ | U+2243 |
| μ | U+03BC | ≈ | U+2248 |
| π | U+03C0 | ≐ | U+2250 |
| ρ | U+03C1 | ≠ | U+2260 |
| σ | U+03C3 | · | U+22C5 |
| ψ | U+03C6 | ⊥ | U+FB29 |
| ψ | U+03C8 | | |

# B. License

```
The MIT License (MIT)

Copyright (c) 2015 Anders Lundstedt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

[13] The Wikipedia page `http://en.wikipedia.org/wiki/Unicode_input` has some information on how to type characters using their Unicode code point.

# C. fin.v—Canonical finite types

`FIN.t n` is a type with `n` members, one member naturally corresponding to each `m < n`.

```
Module FIN.
```

## Definitions

```
  Inductive Singleton (n : nat) := lift : Singleton n.
```

### Canonical finite types

`t 0` is the empty type. `t (S n)` is the sum of `t n` and the singleton of `n`.

```
  Fixpoint t (n : nat) : Type :=
  match n with
  | O   => Empty_set
  | S n => sum (t n) (Singleton n)
  end.

  Definition last {n} : t (S n) := inr (lift n).
```

An element of `FIN.t n` is also an element of `FIN.t (S n)`. More generally, it is also an element of `FIN.t (k + n)`.

```
  Definition up {n} (x : t n) : t (S n) := inl x.
  Local Notation "++ x" := (up x) (at level 6, right associativity).

  Fixpoint up' {n k} (x : t n) : t (k + n) :=
  match k with
  | O   => x
  | S k => ++(up' x)
  end.
```

### To and from naturals

```
  Definition ofNat (n k : nat) : t (k + S n) := up' last.

  Fixpoint toNat {n} : t (S n) -> nat :=
  match n with
  | O   => fun _ => 0
  | S n => fun x => match x with
                    | inl x => toNat x
                    | inr x => n
                    end
  end.
```

## Notations

```
  Module NOTATIONS.

    Delimit Scope FIN with FIN.
```

```
    Notation "++ x" := ++x (at level 6, right associativity) : FIN.

  End NOTATIONS.
```

### Theorems

```
  Module Type THMS_SIG.

    Axiom t0Empty : t 0 -> False.

    Axiom t1Singleton : forall x : t 1, x = last.
```

One would want to prove the correctness of `ofNat` and `toNat`, i.e. that

```
ofNat (toNat x) k = up' x
```

and that

```
toNat (ofNat n k) = n.
```

It is however nontrivial to even state these, since `(k + S n) = (S _)` does not automatically typecheck. For my purposes, I do not need to prove their correctness so I will skip doing so.

```
  End THMS_SIG.
```

### Proofs

```
  Module THMS : THMS_SIG.

    Theorem t0Empty (x : t 0) : False.
    Proof. destruct x. Qed.

    Theorem t1Singleton (x : t 1) : x = last.
    Proof. destruct x as [x | x]; destruct x. reflexivity. Qed.

  End THMS.

End FIN.
```

## D. vec.v—A vector library

I am probably more or less reinventing the wheel here, but I found the standard library's vector implementations (`Coq.Vectors.*`) insufficient for two reasons. The first reason is that vectors are defined as separate inductive types, instead of as functions to `Type`, with which a lot would come for free by computation and with which induction proofs would be easier. The second reason is that all functions are defined under the interpretation that vectors are built by appending a vector to an element. For my purposes the reverse interpretation—that vectors are built by appending an element to a vector—is more suitable.

```
Require Import Coq.Classes.Morphisms.

Require Import fin.

Import FIN.NOTATIONS.
Local Open Scope FIN.

Local Infix "∘" :=
  (fun g f => fun x => g (f x)) (at level 5, left associativity).

Module VEC.
```

## Definitions

### One empty vector for every type

```
  Inductive Empty (A : Type) := empty.
  Arguments empty {_}.
```

### Vectors

For each type `A`, there is one `A`-vector of length `0`—the empty vector—and the type of `A`-vectors of length `S n` is the product of the type of `A`-vectors of length `n` and `A`.

```
  Fixpoint t A (n : nat) : Type :=
  match n with
  | O   => Empty A
  | S n => prod (t A n) A
  end.
  Local Notation "()" := (empty : t _ 0).
  Local Infix ";;" :=
    ((fun A n (v : t A n) (a : A) => ((v, a) : t A (S n))) _ _)
    (at level 40, left associativity).
```

### Accessing elements of a vector

```
  Fixpoint nth {A} {n} : t A n -> FIN.t n -> A :=
  match n with
  | O   => fun _ x => False_rect _ (FIN.THMS.t0Empty x)
  | S n => fun v x => let (v, a) := v in
                      match x with
                      | inl x => nth v x
                      | inr _ => a
                      end
  end.
```

### Vectors of copies of an element

```
  Fixpoint copies {A} (n : nat) (a : A) : t A n :=
  match n with
  | O   => ()
  | S n => (copies n a, a)
  end.
```

### Concatenation of vectors

```
Fixpoint concat {A} {n m} (v : t A n) : t A m -> t A (m + n) :=
match m with
| O   => fun _ => v
| S m => fun w => let (w, a) := w in (concat v w, a)
end.
```

### Mapping functions on vectors

The map of `f : A -> B` over an `A`-vector `((), a1, a2, .., an)` is the `B`-vector `((), f a1, f a2, ..., f an)`.

```
Fixpoint map {A B} {n} (f : A -> B) : t A n -> t B n :=
match n with
| O   => fun _ => ()
| S n => fun v => let (v, a) := v in (map f v, f a)
end.
```

### Vectors as the image of functions on the canonically finite sets

```
Fixpoint finMap {A} {n} : (FIN.t n -> A) -> t A n :=
match n with
| O   => fun _ => ()
| S n => fun f => (finMap (fun x => f ++x), f FIN.last)
end.

Definition finMap' {A} {n k} (f : FIN.t (k + n) -> A) : t A n :=
  finMap (fun x => f (FIN.up' x)).
```

### Left fold

```
Fixpoint foldl {A} {n} (f : A -> A -> A) : A -> t A n -> A :=
match n with
| O   => fun a _ => a
| S n => fun a v => let (v, b) := v in f (foldl f a v) b
end.
```

## Notations

```
Module NOTATIONS.

  Delimit Scope VEC with VEC.

  Notation "()" := () : VEC.
  Infix ";;" := (fun v a => v;; a) (at level 40, left associativity) : VEC.

End NOTATIONS.
```

## Theorems

```
Module Type THMS_SIG.
```

### Correctness of copies

```
Axiom nthCopiesEq :
  forall {A} {n} (a : A) (x : FIN.t n), nth (copies n a) x = a.
```

### Correctness of concatenation

```
Axiom nthConcatEq1 :
  forall  {A} {n m} (x : FIN.t n) (v : t A n) (w : t A m),
  nth (concat v w) (FIN.up' x) = nth v x.
```

It is harder to state that concatenation is correct with respect to the second vector. I do not need this fact so I will skip it.

### Correctness of map

```
Axiom nthMapEq :
  forall {A B} {n} (f : A -> B) (v : t A n) (x : FIN.t n),
  nth (map f v) x = f (nth v x).
```

### Correctness of maps from canonically finite types

```
Axiom nthFinMapEq :
  forall {A} {n} (f : FIN.t n -> A) (x : FIN.t n),
  VEC.nth (finMap f) x = f x.

Axiom nthFinMapEq' :
  forall {A} {n k} (f : FIN.t (k + n) -> A) (x : FIN.t n),
  VEC.nth (finMap' f) x = f (FIN.up' x).
```

### Equality is pointwise equality

```
Axiom pointwiseEquality :
  forall {A} {n} (v w : t A n),
  v = w <-> (forall x : FIN.t n, nth v x = nth w x).
```

### Map respects extensionality

```
Declare Instance mapRespectsExtensionality {A B} {n} :
  Proper ((fun f g => forall a : A, f a = g a) ==> eq ==> eq) (@map A B n).
```

### Map and composition

```
Axiom mapComposeEq :
  forall {A B C} {n} (f : A -> B) (g : B -> C) (v : t A n),
  map g (map f v) = map g∘f v.
```

### Map and concatenation

```
Axiom mapConcatEq :
  forall {A B} {n m} (f : A -> B) (v : VEC.t A n) (w : VEC.t A m),
  map f (concat v w) = concat (map f v) (map f w).
```

### Maps from canonically finite types and composition

```
  Axiom finMapComposeEq :
    forall {A B} {n} (f : FIN.t n -> A) (g : A -> B),
    map g (finMap f) = finMap g∘f.

  Axiom finMapComposeEq' :
    forall {A B} {n k} (f : FIN.t (k+n) -> A) (g : A -> B),
    map g (finMap' f) = finMap' g∘f.
```

### Left fold and concatenation

```
  Axiom foldlConcatEq :
    forall {A} {n m} (f : A -> A -> A) (a : A) (v : t A n) (w : t A m),
    foldl f a (concat v w) = foldl f (foldl f a v) w.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Theorem nthCopiesEq {A} {n} (a : A) (x : FIN.t n) : nth (copies n a) x = a.
  Proof.
    induction n as [ | n IHn]; simpl.
    - inversion x.
    - destruct x as [x | x].
      + apply IHn.
      + reflexivity.
  Qed.

  Theorem nthConcatEq1 {A} {n m} (x : FIN.t n) (v : t A n) (w : t A m) :
    nth (concat v w) (FIN.up' x) = nth v x.
  Proof.
    induction m as [ | m IHm].
    - reflexivity.
    - destruct w as [w a]. apply IHm.
  Qed.

  Theorem nthMapEq {A B} {n} (f : A -> B) (v : t A n) (x : FIN.t n) :
    nth (map f v) x = f (nth v x).
  Proof.
    induction n as [ | n IHn].
    - inversion x.
    - destruct v as [v a]. destruct x as [x | x].
      + apply IHn.
      + reflexivity.
  Qed.

  Theorem nthFinMapEq {A} {n} (f : FIN.t n -> A) (x : FIN.t n) :
    VEC.nth (finMap f) x = f x.
  Proof.
    induction n as [ | n IHn].
    - inversion x.
```

```
  - destruct x as [x | x]; simpl.
    + rewrite IHn. reflexivity.
    + destruct x. reflexivity.
Qed.

Theorem nthFinMapEq' {A} {n k} (f : FIN.t (k + n) -> A) (x : FIN.t n) :
  VEC.nth (finMap' f) x = f (FIN.up' x).
Proof. unfold finMap'. rewrite nthFinMapEq. reflexivity. Qed.

Theorem pointwiseEquality {A} {n} (v w : t A n) :
  v = w <-> (forall x : FIN.t n, nth v x = nth w x).
Proof.
  split; [intro; subst; tauto | ].
  intro H.
  induction n as [ | n IHn].
  - destruct v, w. reflexivity.
  - destruct v as [v a], w as [w b].
    specialize (IHn v w).
    pose proof (H FIN.last) as H'. simpl in H'. rewrite H'. clear H'.
    lapply IHn; [intro; subst; tauto | ]. clear IHn.
    intro x. exact (H ++x).
Qed.

Instance mapRespectsExtensionality {A B} {n} :
  Proper ((fun f g => forall a : A, f a = g a) ==> eq ==> eq) (@map A B n).
Proof.
  intros f g Hfg v' v H. subst.
  induction n as [ | n IHn].
  - reflexivity.
  - destruct v as [v a]. simpl. rewrite IHn, Hfg. reflexivity.
Qed.

Theorem mapComposeEq {A B C} {n} (f : A -> B) (g : B -> C) (v : t A n) :
  map g (map f v) = map g∘f v.
Proof.
  induction n as [ | n IHn].
  - reflexivity.
  - destruct v as [v a]. simpl. rewrite IHn. reflexivity.
Qed.

Theorem mapConcatEq {A B} {n m} (f : A -> B) (v : VEC.t A n)
                    (w : VEC.t A m) :
  map f (concat v w) = concat (map f v) (map f w).
Proof.
  induction m as [ | m IHm]; simpl.
  - reflexivity.
  - destruct w as [w a]. rewrite IHm. reflexivity.
Qed.

Theorem finMapComposeEq {A B} {n} (f : FIN.t n -> A) (g : A -> B) :
  map g (finMap f) = finMap g∘f.
Proof.
  induction n as [ | n IHn].
  - reflexivity.
  - simpl. rewrite IHn. reflexivity.
```

```
    Qed.

    Theorem finMapComposeEq' {A B} {n k} (f : FIN.t (k+n) -> A) (g : A -> B) :
      map g (finMap' f) = finMap' g∘f.
    Proof. unfold finMap'. apply finMapComposeEq. Qed.

    Theorem foldlConcatEq {A} {n m} (f : A -> A -> A) (a : A) (v : t A n)
                          (w : t A m) :
      foldl f a (concat v w) = foldl f (foldl f a v) w.
    Proof.
      induction m as [ | m IHm]; simpl.
      - reflexivity.
      - destruct w as [w b]. rewrite IHm. reflexivity.
    Qed.

  End THMS.

End VEC.
```

# E. heytingarithmetic.v—Heyting arithmetic

```
Require Import fin.
Require Import vec.

Import FIN.NOTATIONS.
Import VEC.NOTATIONS.

Local Open Scope FIN.
Local Open Scope VEC.

Module HA.
```

## Definitions

### Terms

```
  Inductive term (n : nat) :=
  | O'    : term n
  | var   : FIN.t n -> term n
  | S'    : term n -> term n
  | plus' : term n -> term n -> term n
  | mult' : term n -> term n -> term n.
  Arguments O' {_}.
  Arguments S' {_} _.
  Arguments var {_} _.
  Arguments plus' {_} _ _.
  Arguments mult' {_} _ _.
  Local Infix "+" := plus' (at level 64).
  Local Infix "·" := mult' (at level 63).

  Definition x01 : term 1 := var (FIN.ofNat 0 0).
  Definition x02 : term 2 := var (FIN.ofNat 0 1).
  Definition x12 : term 2 := var (FIN.ofNat 1 0).
```

```
Definition x03 : term 3 := var (FIN.ofNat 0 2).
Definition x13 : term 3 := var (FIN.ofNat 1 1).
Definition x23 : term 3 := var (FIN.ofNat 2 0).
Definition x04 : term 4 := var (FIN.ofNat 0 3).
Definition x14 : term 4 := var (FIN.ofNat 1 2).
Definition x24 : term 4 := var (FIN.ofNat 2 1).
Definition x34 : term 4 := var (FIN.ofNat 3 0).
```

### Atomic formulas

```
Inductive atom (n : nat) :=
| FF     : atom n
| atomEq : term n -> term n -> atom n.
Arguments FF {_}.
Arguments atomEq {_} _ _.
Local Infix "≐" := atomEq (at level 65).
```

### Formulas

```
Inductive formula n :=
| fAtom : atom n -> formula n
| fAnd  : formula n -> formula n -> formula n
| fOr   : formula n -> formula n -> formula n
| fImp  : formula n -> formula n -> formula n
| fEx   : formula (S n) -> formula n
| fFa   : formula (S n) -> formula n.
Arguments fAtom {_} _.
Arguments fAnd {_} _ _.
Arguments fOr {_} _ _.
Arguments fImp {_} _ _.
Arguments fEx {_} _.
Arguments fFa {_} _.
Local Coercion fAtom : atom >-> formula.
Local Infix "∧" := fAnd (at level 67, left associativity).
Local Infix "∨" := fOr (at level 68, left associativity).
Local Infix "→" := fImp (at level 69, right associativity).
Local Notation "¬ A" := (A → FF) (at level 66, right associativity).
Local Infix "≠" := (fun u v => ¬(u ≐ v)) (at level 65).
Local Notation "∃ A" := (fEx A) (at level 5).
Local Notation "∀ A" := (fFa A) (at level 5).
```

### Term promotion

```
Fixpoint termUp {n : nat} (t : term n) : term (S n) :=
match t with
| O'    => O'
| var x => var ++x
| S' t  => S' (termUp t)
| u + v => (termUp u) + (termUp v)
| u · v => (termUp u) · (termUp v)
end.

Definition termVecUp {n m} (f : VEC.t (term m) n) : VEC.t (term (S m)) n :=
  VEC.map termUp f.
```

### Identity term vectors

```
Definition idVec {n} : VEC.t (term n) n := VEC.finMap var.

Definition idVecUp {n} : VEC.t (term (S n)) n := VEC.finMap' (k := 1) var.
```

### Substitution in terms and formulas

```
Fixpoint termSubst {n m} (t : term n) (f : VEC.t (term m) n) : term m :=
match t with
| O'    => O'
| var x => VEC.nth f x
| S' t  => S' (termSubst t f)
| u ⊹ v => termSubst u f ⊹ termSubst v f
| u · v => termSubst u f · termSubst v f
end.

Definition atomSubst {n m} (A : atom n) (f : VEC.t (term m) n) : atom m :=
match A with
| FF    => FF
| u ≐ v => termSubst u f ≐ termSubst v f
end.

Fixpoint formulaSubst {n m} (A : formula n) (f : VEC.t (term m) n)
  : formula m :=
match A with
| fAtom A => atomSubst A f
| A ∧ B   => formulaSubst A f ∧ formulaSubst B f
| A ∨ B   => formulaSubst A f ∨ formulaSubst B f
| A → B   => formulaSubst A f → formulaSubst B f
| ∃A      => ∃(formulaSubst A ((termVecUp f), var FIN.last))
| ∀A      => ∀(formulaSubst A ((termVecUp f), var FIN.last))
end.
Local Infix "//" := formulaSubst (at level 62, left associativity).
```

### Substitution in the last variable

```
Definition lastSubst {n} (A : formula (S n)) (t : term (S n)) : formula (S n)
  := A // (idVecUp, t).
Local Infix "/+" := lastSubst (at level 62, left associativity).

Definition downSubst {n} (A : formula (S n)) (t : term n) : formula n :=
  A // (idVec, t).
Local Infix "/-" := downSubst (at level 62, left associativity).
```

### Formula promotion

```
Definition formulaUp {n} (A : formula n) : formula (S n) := A // idVecUp.
Local Notation up := formulaUp.
```

### Semantics

```
Class Interpretation (Truth : forall {n}, formula n -> Prop) := {
  Consistency : ~Truth (@FF 0);
```

```
  RuleMP {n} (A B : formula n) : Truth (A → B) -> Truth A -> Truth B;
  RuleGen {n} (A : formula (S n)) : Truth A -> Truth ∀A;
  RuleUp {n} (A : formula n) : Truth A -> Truth (up A);
  AxImp1 {n} (A B : formula n) : Truth (A → B → A);
  AxImp2 {n} (A B C : formula n) : Truth ((A → B → C) → (A → B) → A → C);
  AxAndI {n} (A B : formula n) : Truth (A → B → A ∧ B);
  AxAndE1 {n} (A B : formula n) : Truth (A ∧ B → A);
  AxAndE2 {n} (A B : formula n) : Truth (A ∧ B → B);
  AxOrI1 {n} (A B : formula n) : Truth (A → A ∨ B);
  AxOrI2 {n} (A B : formula n) : Truth (B → A ∨ B);
  AxOrE {n} (A B C : formula n) : Truth (A ∨ B → (A → C) → (B → C) → C);
  AxExFalso {n} (A : formula n) : Truth (FF → A);
  AxExI {n} (A : formula (S n)) (t : term n) : Truth (A/-t → ∃A);
  AxExE {n} (A : formula (S n)) (B : formula n) :
    Truth (∀(A → up B) → ∃A → B);
  AxFaI {n} (A : formula n) (B : formula (S n)) :
    Truth (∀(up A → B) → A → ∀B);
  AxFaE {n} (A : formula (S n)) (t : term n) : Truth (∀A → A/-t);
  AxEqRefl : Truth (x01 ≐ x01);
  AxEqSymm : Truth (x02 ≐ x12 → x12 ≐ x02);
  AxEqTrans : Truth (x03 ≐ x13 → x13 ≐ x23 → x03 ≐ x23);
  AxEqS : Truth (x02 ≐ x12 → S' x02 ≐ S' x12);
  AxEqPlus : Truth (x04 ≐ x14 → x24 ≐ x34 → x04 + x24 ≐ x14 + x34);
  AxEqMult : Truth (x04 ≐ x14 → x24 ≐ x34 → x04 · x24 ≐ x14 · x34);
  AxPA1 : Truth (S' x01 ≠ x01);
  AxPA2 : Truth (S' x02 ≐ S' x12 → x02 ≐ x12);
  AxPA3 : Truth (x01 + O' ≐ x01);
  AxPA4 : Truth (x02 + S' x12 ≐ S' (x02 + x12));
  AxPA5 : Truth (x01 · O' ≐ O');
  AxPA6 : Truth (x02 · S' x12 ≐ x02·x12 + x02);
  AxPA7 {n} (A : formula (S n)) :
    Truth (A/-O' → ∀(A → A/+(S' (var FIN.last))) → ∀A)
}.
```

## Notations

```
Module NOTATIONS.

  Delimit Scope HA with HA.

  Infix "+" := plus' (at level 64) : HA.
  Infix "·" := mult' (at level 63) : HA.
  Infix "≐" := atomEq (at level 65) : HA.
  Notation "¬ A" := (¬A) (at level 66, right associativity) : HA.
  Infix "≠" := (fun u v => u ≠ v) (at level 65) : HA.
  Infix "∧" := fAnd (at level 67, left associativity) : HA.
  Infix "∨" := fOr (at level 68, left associativity) : HA.
  Infix "→" := fImp (at level 69, right associativity) : HA.
  Notation "∃ A" := ∃A (at level 5) : HA.
  Notation "∀ A" := ∀A (at level 5) : HA.
  Infix "//" := (fun A f => A//f) (at level 62, left associativity) : HA.
  Infix "/+" := (fun A t => A/+t) (at level 62, left associativity) : HA.
  Infix "/-" := (fun A t => A/-t) (at level 62, left associativity) : HA.
  Notation up := (fun A => up A).
```

```
  End NOTATIONS.

End HA.
```

## Coercions

```
Coercion HA.fAtom : HA.atom >-> HA.formula.
```

# F. standardha.v—Standard interpretation of Heyting arithmetic

```
Require Import Coq.Setoids.Setoid.

Require Import fin.
Require Import heytingarithmetic.
Require Import vec.

Import FIN.NOTATIONS.
Import HA.
Import HA.NOTATIONS.
Import VEC.NOTATIONS.

Local Open Scope FIN.
Local Open Scope VEC.
Local Open Scope HA.

Module STD_HA.
```

## Definitions

### Term valuation

```
  Fixpoint termVal {n} (t : term n) : VEC.t nat n -> nat :=
  match t with
  | O'     => fun _ => 0
  | var x  => fun f => VEC.nth f x
  | S' t   => fun f => S (termVal t f)
  | u + v  => fun f => termVal u f + termVal v f
  | u · v  => fun f => termVal u f * termVal v f
  end.

  Definition termVecVal {n m} (f : VEC.t (term m) n) (g : VEC.t nat m) :
    VEC.t nat n := VEC.map (fun t => termVal t g) f.
```

### Atomic truth

```
  Definition AtomicTruthPred {n} (A : atom n) : VEC.t nat n -> Prop :=
  match A with
  | FF    => fun _ => False
  | u ≐ v => fun f => (termVal u f) = (termVal v f)
  end.
```

### Truth

```
Fixpoint TruthPred {n} (A : formula n) : VEC.t nat n -> Prop :=
match A with
| fAtom A => AtomicTruthPred A
| A ∧ B   => fun f => TruthPred A f /\ TruthPred B f
| A ∨ B   => fun f => TruthPred A f \/ TruthPred B f
| A → B   => fun f => TruthPred A f -> TruthPred B f
| ∃A      => fun f => exists x : nat, TruthPred A (f, x)
| ∀A      => fun f => forall x : nat, TruthPred A (f, x)
end.

Definition Truth {n} (A : formula n) : Prop :=
  forall f : VEC.t nat n, TruthPred A f.
```

## Theorems

```
Module Type THMS_SIG.

  Declare Instance interpretation : Interpretation (@Truth).

  Axiom termVecValIdVecEq :
    forall {n} (f : VEC.t nat n), termVecVal idVec f = f.

  Axiom termVecValIdVecUpEq :
    forall {n} (f : VEC.t nat n) (x : nat), termVecVal idVecUp (f, x) = f.

  Axiom termValTermUpEq :
    forall {n} (t : term n) (f : VEC.t nat n) (x : nat),
    termVal (termUp t) (f, x) = termVal t f.

  Axiom termVecValTermVecUpEq :
    forall {n m} (f : VEC.t (term m) n) (g : VEC.t nat m) (x : nat),
    termVecVal (termVecUp f) (g, x) = termVecVal f g.

  Axiom termVecValTermVecUpEq' :
    forall {n m} (f : VEC.t (term m) n) (g : VEC.t nat m) (x : nat),
    termVecVal (termVecUp f;; var FIN.last) (g;; x) = (termVecVal f g;; x).

  Axiom termValTermSubstEq :
    forall {n m} (t : term n) (f : VEC.t (term m) n) (g : VEC.t nat m),
    termVal (termSubst t f) g = termVal t (termVecVal f g).

  Axiom atomicTruthSubstEq :
    forall {n m} (A : atom n) (f : VEC.t (term m) n) (g : VEC.t nat m),
    AtomicTruthPred (atomSubst A f) g <->
    AtomicTruthPred A (termVecVal f g).

  Axiom truthSubstEq :
    forall {n m} (A : formula n) (f : VEC.t (term m) n) (g : VEC.t nat m),
    TruthPred (A // f) g <-> TruthPred A (termVecVal f g).

  Axiom truthLastSubstEq :
    forall {n} (A : formula (S n)) (t : term (S n)) (f : VEC.t nat n)
           (x : nat),
```

```
    TruthPred (A /+ t) (f, x) <-> TruthPred A (f, termVal t (f, x)).

  Axiom truthDownSubstEq :
    forall {n} (A : formula (S n)) (t : term n) (f : VEC.t nat n),
    TruthPred (A /- t) f <-> TruthPred A (f, termVal t f).

  Axiom truthUpEq :
    forall {n} (A : formula n) (f : VEC.t nat n) (x : nat),
    TruthPred (up A) (f, x) <-> TruthPred A f.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Theorem termVecValIdVecEq {n} (f : VEC.t nat n) :
    termVecVal idVec f = f.
  Proof.
    unfold idVec, termVecVal. rewrite VEC.THMS.finMapComposeEq. simpl.
    apply VEC.THMS.pointwiseEquality, VEC.THMS.nthFinMapEq.
  Qed.

  Theorem termVecValIdVecUpEq {n} (f : VEC.t nat n) (x : nat) :
    termVecVal idVecUp (f, x) = f.
  Proof.
    unfold idVecUp, termVecVal. rewrite VEC.THMS.finMapComposeEq'. simpl.
    apply VEC.THMS.pointwiseEquality, VEC.THMS.nthFinMapEq.
  Qed.

  Theorem termValTermUpEq {n} (t : term n) (f : VEC.t nat n) (x : nat) :
    termVal (termUp t) (f, x) = termVal t f.
  Proof.
    induction t as [a | y | t IH | u IHu v IHv | u IHu v IHv]; simpl.
    - reflexivity.
    - reflexivity.
    - rewrite IH. reflexivity.
    - rewrite IHu, IHv. reflexivity.
    - rewrite IHu, IHv. reflexivity.
  Qed.

  Theorem termVecValTermVecUpEq {n m} (f : VEC.t (term m) n) (g : VEC.t nat m)
                                (x : nat) :
    termVecVal (termVecUp f) (g, x) = termVecVal f g.
  Proof.
    setoid_rewrite VEC.THMS.mapComposeEq.
    apply VEC.THMS.mapRespectsExtensionality; [ | reflexivity].
    intro t. apply termValTermUpEq.
  Qed.

  Theorem termVecValTermVecUpEq' {n m} (f : VEC.t (term m) n)
                                 (g : VEC.t nat m) (x : nat) :
    termVecVal (termVecUp f;; var FIN.last) (g;; x) = (termVecVal f g;; x).
  Proof.
    assert
```

```
    (termVecVal (termVecUp f;; var FIN.last) (g;; x) =
     (termVecVal (termVecUp f) (g;; x), x))
  as H by reflexivity.
  setoid_rewrite H. clear H.
  rewrite termVecValTermVecUpEq.
  reflexivity.
Qed.

Theorem termValTermSubstEq {n m} (t : term n) (f : VEC.t (term m) n)
                           (g : VEC.t nat m) :
  termVal (termSubst t f) g = termVal t (termVecVal f g).
Proof.
  induction t as [a | x | t IH | u IHu v IHv | u IHu v IHv]; simpl.
  - reflexivity.
  - unfold termVecVal. rewrite VEC.THMS.nthMapEq. reflexivity.
  - rewrite IH. reflexivity.
  - rewrite IHu, IHv. reflexivity.
  - rewrite IHu, IHv. reflexivity.
Qed.

Theorem atomicTruthSubstEq {n m} (A : atom n) (f : VEC.t (term m) n)
                           (g : VEC.t nat m) :
  AtomicTruthPred (atomSubst A f) g <-> AtomicTruthPred A (termVecVal f g).
Proof.
  destruct A as [ | u v]; simpl.
  - tauto.
  - setoid_rewrite termValTermSubstEq. tauto.
Qed.

Theorem truthSubstEq {n m} (A : formula n) (f : VEC.t (term m) n)
                     (g : VEC.t nat m) :
  TruthPred (A // f) g <-> TruthPred A (termVecVal f g).
Proof.
  generalize dependent m.
  induction A as
    [n A | n A IHA B IHB | n A IHA B IHB | n A IHA B IHB | n A IH | n A IH];
    intros m f g; simpl.
  - apply atomicTruthSubstEq.
  - rewrite IHA, IHB. tauto.
  - rewrite IHA, IHB. tauto.
  - rewrite IHA, IHB. tauto.
  - setoid_rewrite IH. setoid_rewrite termVecValTermVecUpEq'. tauto.
  - setoid_rewrite IH. setoid_rewrite termVecValTermVecUpEq'. tauto.
Qed.

Theorem truthLastSubstEq {n} (A : formula (S n)) (t : term (S n))
                         (f : VEC.t nat n) (x : nat) :
  TruthPred (A /+ t) (f, x) <-> TruthPred A (f, termVal t (f, x)).
Proof.
  setoid_rewrite truthSubstEq.
  assert
    (termVecVal (idVecUp;; t) (f;; x) =
     (termVecVal idVecUp (f;; x));; termVal t (f;; x))
  as H by reflexivity.
  setoid_rewrite H. clear H.
```

```
  rewrite termVecValIdVecUpEq.
  tauto.
Qed.

Theorem truthDownSubstEq {n} (A : formula (S n)) (t : term n)
                                  (f : VEC.t nat n) :
  TruthPred (A /- t) f <-> TruthPred A (f, termVal t f).
Proof.
  setoid_rewrite truthSubstEq.
  assert (termVecVal (idVec;; t) f = (termVecVal idVec f, termVal t f))
    as H by reflexivity.
  setoid_rewrite H. clear H.
  rewrite termVecValIdVecEq.
  tauto.
Qed.

Theorem truthUpEq {n} (A : formula n) (f : VEC.t nat n) (x : nat) :
  TruthPred (up A) (f, x) <-> TruthPred A f.
Proof.
  unfold formulaUp. rewrite truthSubstEq, termVecValIdVecUpEq. tauto.
Qed.

Instance interpretation : Interpretation (@Truth).
Proof.
  split; unfold Truth; simpl.
  - intro H. exact (H ()).
  - firstorder.
  - firstorder.
  - intros n A H [f x]. apply truthUpEq. apply H.
  - tauto.
  - tauto.
  - tauto.
  - tauto.
  - tauto.
  - tauto.
  - tauto.
  - tauto.
  - tauto.
  - setoid_rewrite truthDownSubstEq. eauto.
  - setoid_rewrite truthUpEq. firstorder.
  - setoid_rewrite truthUpEq. firstorder.
  - setoid_rewrite truthDownSubstEq. auto.
  - congruence.
  - congruence.
  - congruence.
  - congruence.
  - congruence.
  - congruence.
  - intros [_ x] H. eapply n_Sn. eauto.
  - firstorder.
  - firstorder.
  - firstorder.
  - firstorder.
  - firstorder.
  - intros n A f H IH.
```

```
        rewrite truthDownSubstEq in H.
        setoid_rewrite truthLastSubstEq in IH.
        induction x as [ | x IHx].
        + exact H.
        + exact (IH x IHx).
    Qed.

  End THMS.

End STD_HA.
```

# G. pas.v—Partial applicative structures

```
Require Import Coq.Classes.Morphisms.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Setoids.Setoid.
Local Notation equiv := SetoidClass.equiv.

Require Import fin.
Require Import vec.

Import FIN.NOTATIONS.
Import VEC.NOTATIONS.

Local Open Scope VEC.

Module PAS.
```

## Definitions

### Partial applicative structure

For generality, the carrier is a setoid and application is a ternary relation that is functional with respect to the setoid equality.

```
  Local Generalizable Variables carrier.
  Local Generalizable Variables setoid.

  Class Pas `(setoid : Setoid carrier)
            (Appl : carrier -> carrier -> carrier -> Prop) := {
    applRespectful (a a' b b' c c' : carrier) :
      a == a' -> b == b' -> c == c' -> Appl a b c -> Appl a' b' c';
    applFunctional (a b c d : carrier) :  Appl a b c -> Appl a b d -> c == d
  }.

  Definition Total `(_ : Pas) : Prop :=
    forall a b : carrier, exists c : carrier, Appl a b c.
```

### Terms

```
  Inductive term `{_ : Pas} (n : nat) :=
  | const : carrier -> term n
  | var   : FIN.t n -> term n
```

```
| appl  : term n -> term n -> term n.
Arguments const {_} {_} {_} {_} {_} _.
Arguments var {_} {_} {_} {_} {_} _.
Arguments appl {_} {_} {_} {_} {_} _ _.
Local Notation "& t" := (const t) (at level 6).
Local Infix "*" := appl (at level 40, left associativity).

Definition lastVar `{_ : Pas} {n} : term (S n) := var FIN.last.

Definition x01 `{_ : Pas} : term 1 := var (FIN.ofNat 0 0).
Definition x02 `{_ : Pas} : term 2 := var (FIN.ofNat 0 1).
Definition x12 `{_ : Pas} : term 2 := var (FIN.ofNat 1 0).
Definition x03 `{_ : Pas} : term 3 := var (FIN.ofNat 0 2).
Definition x13 `{_ : Pas} : term 3 := var (FIN.ofNat 1 1).
Definition x23 `{_ : Pas} : term 3 := var (FIN.ofNat 2 0).
Definition x04 `{_ : Pas} : term 4 := var (FIN.ofNat 0 3).
Definition x14 `{_ : Pas} : term 4 := var (FIN.ofNat 1 2).
Definition x24 `{_ : Pas} : term 4 := var (FIN.ofNat 2 1).
Definition x34 `{_ : Pas} : term 4 := var (FIN.ofNat 3 0).
Definition x05 `{_ : Pas} : term 5 := var (FIN.ofNat 0 4).
Definition x15 `{_ : Pas} : term 5 := var (FIN.ofNat 1 3).
Definition x25 `{_ : Pas} : term 5 := var (FIN.ofNat 2 2).
Definition x35 `{_ : Pas} : term 5 := var (FIN.ofNat 3 1).
Definition x45 `{_ : Pas} : term 5 := var (FIN.ofNat 4 0).

Fixpoint closedTermToTerm `{_ : Pas} {n} (t : term 0) : term n :=
match t with
| &a    => &a
| var x => False_rect _ (FIN.THMS.t0Empty x)
| u*v   => closedTermToTerm u * closedTermToTerm v
end.
Local Notation "§ t" := (closedTermToTerm t) (at level 6).
```

## Vectors of terms

```
Definition pasVecToTermVec `{pas : Pas} {m n : nat} (f : VEC.t carrier n)
  : VEC.t (term m) n := VEC.map const f.
Local Notation "&& P" := (pasVecToTermVec P) (at level 6).

Definition idVec `{_ : Pas} {n} k : VEC.t (term (k + n)) n := VEC.finMap' var.
```

## Subterms

```
Inductive Subterm `{_ : Pas} {n} : term n -> term n -> Prop :=
| subtermRefl u     : Subterm u u
| subtermL    u v w : Subterm u v -> Subterm u (v*w)
| subtermR    u v w : Subterm u w -> Subterm u (v*w).
Hint Constructors Subterm.
```

## Substitution

```
Fixpoint substitution `{_ : Pas} {m n : nat} (t : term n)
                      (f : VEC.t (term m) n) : term m :=
match t with
```

```
| &a    => &a
| var x => VEC.nth f x
| u*v   => substitution u f * substitution v f
end.
Local Infix "/" := substitution.

Definition termVecSubst `{_ : Pas} {n m k} (f : VEC.t (term m) n)
                        (g : VEC.t (term k) m) : VEC.t (term k) n :=
  VEC.map (fun t => t/g) f.
```

### Product of a term and a vector of terms

```
Definition product `{_ : Pas} {m n : nat} (t : term m) (f : VEC.t (term m) n)
  : term m := VEC.foldl appl t f.
Local Infix "**" := product (at level 39, left associativity).
```

### Equivalence of closed terms

```
Fixpoint Denotation `{pas : Pas} (t : term 0) (c : carrier) : Prop :=
match t with
| &a    => a == c
| u*v   => exists a b, Denotation u a /\ Denotation v b /\ Appl a b c
| var _ => False
end.

Definition ClosedTermEq `{_ : Pas} (u v : term 0) : Prop :=
  forall a, Denotation u a <-> Denotation v a.
Local Infix "≃" := ClosedTermEq (at level 70).
```

### Denoting closed terms

```
Definition Denotes `{_ : Pas} (t : term 0) : Prop :=
  exists a : carrier, t ≃ &a.
Hint Unfold Denotes.
Local Notation "t !" := (Denotes t) (at level 70).
```

## Notations

```
Module NOTATIONS.

  Delimit Scope PAS with PAS.

  Notation "§ t" := § t (at level 6) : PAS.
  Notation "& t" := &t (at level 6) : PAS.
  Notation "&& f" := &&f (at level 6) : PAS.
  Notation "t !" := (t!) (at level 70) : PAS.
  Infix "*" := appl : PAS.
  Infix "**" := product (at level 39, left associativity) : PAS.
  Infix "/" := substitution : PAS.
  Infix "≃" := ClosedTermEq (at level 70) : PAS.

End NOTATIONS.
```

## Theorems

```
Module Type THMS_SIG.
```

### Application respects setoid equality

```
Declare Instance applRespectful `{_ : Pas} :
  Proper (equiv ==> equiv ==> equiv ==> iff) Appl.
```

### Closed term equivalence correctly implemented

```
Declare Instance closedTermEqEquivalence `{_ : Pas} :
  Equivalence ClosedTermEq.

Declare Instance constRespectsClosedTermEq `{_ : Pas} :
  Proper (equiv ==> ClosedTermEq) const.

Declare Instance applRespectsClosedTermEq `{_ : Pas} :
  Proper (ClosedTermEq ==> ClosedTermEq ==> ClosedTermEq) appl.

Axiom constInjective : forall `{_ : Pas} (a b : carrier), &a ≃ &b -> a == b.

Axiom correctness1 :
  forall `{_ : Pas} (u v : term 0) (c : carrier),
  u*v ≃ &c <-> exists a b : carrier, u ≃ &a /\ v ≃ &b /\ Appl a b c.

Axiom correctness1' :
  forall `{_ : Pas} (a b c : carrier), &a*&b ≃ &c <-> Appl a b c.

Axiom correctness2 :
  forall `{_ : Pas} (u v : term 0),
  u ≃ v <-> forall a : carrier, u ≃ &a <-> v ≃ &a.
```

### Denotation respects closed term equivalence

```
Declare Instance denotationRespectsClosedTermEq `{_ : Pas} :
  Proper (ClosedTermEq ==> iff) Denotes.
```

### Closed term to closed term is the identity

```
Axiom closedTermToClosedTermEq : forall `{_ : Pas} (t : term 0), $t = t.
```

### Substitution of identity vector is the identity

```
Axiom substIdVecEq : forall `{_ : Pas} {n} (t : term n), t / idVec 0 = t.
```

### Substitution in closed terms is the identity

```
Axiom closedTermSubstitutionEq :
  forall `{_ : Pas} {n m : nat} (t : term 0) (f : VEC.t (term m) n),
  $t/f = $t.
```

```
Axiom closedTermSubstitutionEq0 :
  forall `{_ : Pas} {n : nat} (t : term 0) (f : VEC.t (term 0) n), $t/f = t.

Axiom emptySubstitutionEq :
  forall `{_ : Pas} {n} (t : term 0) (f : VEC.t (term n) 0), t/f = $t.
```

### Substitution in identity vector

```
Axiom termVecSubstIdVecEq :
  forall {n m k} `{_ : Pas} (f : VEC.t (term m) n) (g : VEC.t (term m) k),
  termVecSubst (idVec k) (VEC.concat f g) = f.
```

### Product of a term and two term vectors

```
Axiom doubleProductEq :
  forall `{_ : Pas} {n m k : nat} (t : term n) (f : VEC.t (term n) m)
         (g : VEC.t (term n) k),
  t ** f ** g = t ** (VEC.concat f g).
```

### Substitution and product of a term and vector

```
Axiom productSubstitutionEq :
  forall `{_ : Pas} {n m k} (t : term m) (f : VEC.t (term m) n)
         (g : VEC.t (term k) m),
  (t ** f) / g = (t/g) ** (termVecSubst f g).

Axiom idVecProductSubstitutionEq :
  forall `{_ : Pas} {n m k} (t : term (k + n)) (f : VEC.t (term m) n)
         (g : VEC.t (term m) k),
  (t ** idVec k) / VEC.concat f g = (t / VEC.concat f g) ** f.

Axiom idVecProductSubstitutionEq0 :
  forall `{_ : Pas} {n m} (t : term n) (f : VEC.t (term m) n),
  (t ** idVec 0) / f = (t/f) ** f.

Axiom idVecProductSubstitutionEq0' :
  forall `{_ : Pas} {n} (t : term 0) (f : VEC.t (term 0) n),
  ($t ** idVec 0) / f = t ** f.

Axiom idVecProductSubstitutionEq1 :
  forall `{_ : Pas} {n m} (u : term (S n)) (f : VEC.t (term m) n)
         (v : term m),
  (u ** idVec 1) / (f;; v) = (u / (f;; v)) ** f.

Axiom idVecProductSubstitutionEq1' :
  forall `{_ : Pas} {n} (u v : term 0) (f : VEC.t (term 0) n),
  ($u ** idVec 1) / (f;; v) = u ** f.
```

### Subterms

```
Declare Instance subtermReflexive `{_ : Pas} {n} :
  Reflexive (Subterm (n := n)).

Declare Instance subtermTransitive `{_ : Pas} {n} :
```

```
    Transitive (Subterm (n := n)).

  Axiom subtermDenotes :
    forall `{_ : Pas} (u v : term 0), Subterm u v -> v! -> u!.

  Axiom subtermDenotesProduct :
    forall `{_ : Pas} {n : nat} (t : term 0) (f : VEC.t (term 0) n),
    t**f! -> t!.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Instance applRespectful `{pas : Pas} :
    Proper (equiv ==> equiv ==> equiv ==> iff) Appl.
  Proof.
    intros a a' Ha b b' Hb c c' Hc.
    split; [ | symmetry in Ha, Hb, Hc]; eauto using applRespectful.
  Qed.

  Instance closedTermEqEquivalence `{pas : Pas} : Equivalence ClosedTermEq.
  Proof. firstorder. Qed.

  Lemma denotationsUnique `{pas : Pas} (t : term 0) (a b : carrier) :
    Denotation t a -> Denotation t b -> a == b.
  Proof.
    generalize a, b. clear.
    induction t as [a' | x | u IHu v IHv]; simpl.
    - intros a b Ha Hb. rewrite <- Ha, Hb. reflexivity.
    - contradiction.
    - intros c c' [a [b [Hua [Hvb Habc]]]] [a' [b' [Hua' [Hvb' Habc']]]].
      specialize (IHu a a' Hua Hua').
      specialize (IHv b b' Hvb Hvb').
      rewrite IHu, IHv in Habc. apply (applFunctional a' b'); assumption.
  Qed.

  Lemma denotationRespectful `{pas : Pas} (u v : term 0) (a b : carrier)
    : u ≃ v -> a == b -> Denotation u a -> Denotation v b.
  Proof.
    intros Huv Hab Hua.
    specialize (Huv a). destruct Huv as [Huv _]. specialize (Huv Hua).
    clear dependent u.
    destruct v as [c | | u v]; simpl in *.
    - rewrite Huv, Hab. reflexivity.
    - contradiction.
    - destruct Huv as [a' [b' [Hua' [Hvb' H]]]]. setoid_rewrite <- Hab. eauto.
  Qed.

  Instance denotationProper `{pas : Pas} :
    Proper (ClosedTermEq ==> equiv ==> iff) Denotation.
  Proof.
    intros u v Huv a b Hab. split; intro H.
    - apply (denotationRespectful u v a b); assumption.
```

```
  - apply (denotationRespectful v u b a);
    [symmetry | symmetry | ]; assumption.
Qed.

Lemma denotationClosedTermEq `{pas : Pas} (t : term 0) (a : carrier) :
  t ≃ &a <-> Denotation t a.
Proof.
  split; intro H.
  - apply H. simpl. reflexivity.
  - intro a'. split; intro H'.
    + apply (denotationsUnique t); assumption.
    + assert (a == a') as Ha. {
        apply (denotationsUnique &a); [simpl; reflexivity | exact H'].
      }
      rewrite <- Ha. exact H.
Qed.

Instance applRespectsClosedTermEq `{pas : Pas} :
  Proper (ClosedTermEq ==> ClosedTermEq ==> ClosedTermEq) appl.
Proof.
  intros u u' Hu v v' Hv c. simpl. setoid_rewrite Hu. setoid_rewrite Hv.
  tauto.
Qed.

Instance constRespectsClosedTermEq `{_ : Pas} :
  Proper (equiv ==> ClosedTermEq) const.
Proof. intros a b Hab c. simpl. rewrite Hab. tauto. Qed.

Theorem constInjective `{pas : Pas} (a b : carrier) : &a ≃ &b -> a == b.
Proof. intro H. specialize (H b). simpl in H. apply H. reflexivity. Qed.

Theorem correctness1 `{pas : Pas} (u v : term 0) (c : carrier) :
  u*v ≃ &c <-> exists a b : carrier, u ≃ &a /\ v ≃ &b /\ Appl a b c.
Proof.
  split.
  - intro H. specialize (H c). destruct H as [_ H]. simpl in H.
    lapply H; [clear H; intros [a [b [Hua [Hvb Habc]]]] | reflexivity].
    exists a, b. split; [ | split].
    + apply denotationClosedTermEq. exact Hua.
    + apply denotationClosedTermEq. exact Hvb.
    + exact Habc.
  - intros [a [b [Hua [Hvb Habc]]]].
    apply denotationClosedTermEq. simpl.
    rewrite denotationClosedTermEq in Hua, Hvb.
    eauto.
Qed.

Theorem correctness1' `{pas : Pas} (a b c : carrier) :
  &a*&b ≃ &c <-> Appl a b c.
Proof.
  rewrite correctness1. split.
  - intros [a' [b' [Ha [Hb H]]]].
    apply constInjective in Ha. apply constInjective in Hb.
    rewrite Ha, Hb. exact H.
  - intro H. exists a, b. firstorder.
```

```
Qed.

Theorem correctness2 `{pas : Pas} (u v : term 0) :
  u ≃ v <-> forall a : carrier, u ≃ &a <-> v ≃ &a.
Proof.
  split; intro H.
  - setoid_rewrite H. tauto.
  - setoid_rewrite denotationClosedTermEq in H. exact H.
Qed.

Theorem closedTermToClosedTermEq `{pas : Pas} (t : term 0) : §t = t.
Proof.
  induction t as [a | x | u IHu v IHv]; simpl.
  - reflexivity.
  - inversion x.
  - rewrite IHu, IHv. reflexivity.
Qed.

Instance denotationRespectsClosedTermEq `{pas : Pas} :
  Proper (ClosedTermEq ==> iff) Denotes.
Proof. intros u v H. unfold Denotes. setoid_rewrite H. tauto. Qed.

Theorem substIdVecEq `{pas : Pas} {n} (t : term n)
  : t / idVec 0 = t.
Proof.
  induction t as [a | x | u IHu v IHv].
  - reflexivity.
  - unfold idVec, substitution. rewrite VEC.THMS.nthFinMapEq'. reflexivity.
  - simpl. setoid_rewrite IHu. setoid_rewrite IHv. reflexivity.
Qed.

Theorem closedTermSubstitutionEq `{pas : Pas} {n m : nat} (t : term 0)
                                 (f : VEC.t (term m) n) :
  §t/f = §t.
Proof.
  induction t as [a | x | u IHu v IHv]; simpl.
  - reflexivity.
  - inversion x.
  - rewrite IHu, IHv. reflexivity.
Qed.

Theorem closedTermSubstitutionEq0 `{pas : Pas} {n : nat} (t : term 0)
                                 (f : VEC.t (term 0) n) :
  §t/f = t.
Proof. rewrite closedTermSubstitutionEq. apply closedTermToClosedTermEq. Qed.

Theorem emptySubstitutionEq `{pas : Pas} {n} (t : term 0)
                            (f : VEC.t (term n) 0) :
  t/f = §t.
Proof.
  rewrite <- closedTermToClosedTermEq at 1. apply closedTermSubstitutionEq.
Qed.

Theorem termVecSubstIdVecEq {n m k} `{pas : Pas} (f : VEC.t (term m) n)
                            (g : VEC.t (term m) k) :
```

```
  termVecSubst (idVec k) (VEC.concat f g) = f.
Proof.
  apply VEC.THMS.pointwiseEquality. intro x.
  unfold termVecSubst. rewrite VEC.THMS.nthMapEq.
  unfold idVec. rewrite VEC.THMS.nthFinMapEq'.
  simpl.
  induction k as [ | k IHk].
  - reflexivity.
  - destruct g as [g t]. simpl. apply IHk.
Qed.

Theorem doubleProductEq `{pas : Pas} {n m k : nat} (t : term n)
                        (f : VEC.t (term n) m) (g : VEC.t (term n) k) :
  t ** f ** g = t ** (VEC.concat f g).
Proof. unfold product. rewrite VEC.THMS.foldlConcatEq. reflexivity. Qed.

Theorem productSubstitutionEq `{pas : Pas} {n m k} (t : term m)
                              (f : VEC.t (term m) n)
                              (g : VEC.t (term k) m) :
  (t ** f) / g = (t/g) ** (termVecSubst f g).
Proof.
  induction n as [ | n IHn].
  - reflexivity.
  - destruct f as [f u].
    assert ((t ** (f;; u) /g = ((t ** f) / g) * (u / g)))
      as H by reflexivity.
    rewrite H, IHn. reflexivity.
Qed.

Theorem idVecProductSubstitutionEq `{pas : Pas} {n m k} (t : term (k + n))
                                   (f : VEC.t (term m) n)
                                   (g : VEC.t (term m) k) :
  (t ** idVec k) / VEC.concat f g = (t / VEC.concat f g) ** f.
Proof. rewrite productSubstitutionEq, termVecSubstIdVecEq. reflexivity. Qed.

Theorem idVecProductSubstitutionEq0 `{pas : Pas} {n m} (t : term n)
                                    (f : VEC.t (term m) n) :
  (t ** idVec 0) / f = (t/f) ** f.
Proof. apply (idVecProductSubstitutionEq (t : term (0 + n)) f ()). Qed.

Theorem idVecProductSubstitutionEq0' `{pas : Pas} {n} (t : term 0)
                                     (f : VEC.t (term 0) n) :
  ($t ** idVec 0) / f = t ** f.
Proof.
  rewrite
    (idVecProductSubstitutionEq0 (n := n) (m := 0)),
    closedTermSubstitutionEq, closedTermToClosedTermEq.
  reflexivity.
Qed.

Theorem idVecProductSubstitutionEq1 `{pas : Pas} {n m} (u : term (S n))
                                    (f : VEC.t (term m) n) (v : term m) :
  (u ** idVec 1) / (f;; v) = (u / (f;; v)) ** f.
Proof.
  assert ((f;; v) = VEC.concat f (();; v)) as H by reflexivity.
```

```
    setoid_rewrite H. clear H.
    apply (idVecProductSubstitutionEq (u : term (1 + n))).
  Qed.

  Theorem idVecProductSubstitutionEq1' `{pas : Pas} {n} (u v : term 0)
                                       (f : VEC.t (term 0) n) :
    ($u ** idVec 1) / (f;; v) = u ** f.
  Proof.
    assert ((f;; v) = VEC.concat f (();; v)) as H by reflexivity.
    setoid_rewrite H. clear H.
    rewrite
      idVecProductSubstitutionEq, closedTermSubstitutionEq,
      closedTermToClosedTermEq.
    reflexivity.
  Qed.

  Instance subtermReflexive `{pas : Pas} {n} : Reflexive (Subterm (n := n)).
  Proof. auto. Qed.

  Instance subtermTransitive `{pas : Pas} {n} : Transitive (Subterm (n := n)).
  Proof. intros u v w Huv Hvw. induction Hvw; auto. Qed.

  Theorem subtermDenotes `{pas : Pas} (u v : term 0) :
    Subterm u v -> v! -> u!.
  Proof.
    intros H. induction H as [ | u v w H IH | u v w H IH].
    - tauto.
    - unfold Denotes. setoid_rewrite correctness1. firstorder.
    - unfold Denotes. setoid_rewrite correctness1. firstorder.
  Qed.

  Theorem subtermDenotesProduct `{pas : Pas} {n : nat} (t : term 0)
                                (f : VEC.t (term 0) n) :
    t**f! -> t!.
  Proof.
    induction n as [ | n IHn].
    - destruct f. tauto.
    - destruct f as [f a]. intro H. apply (IHn f).
      apply subtermDenotes with (v := (t ** (f;; a))); [ | exact H].
      apply subtermL. reflexivity.
  Qed.

End THMS.
```

Hide the implementation of closed term equivalence, since only its correctness should matter.

```
  Global Opaque Denotation ClosedTermEq.

End PAS.
```

# H. pca.v—Partial combinatory algebras

```
Require Coq.Arith.Minus.
Require Coq.Numbers.Natural.Peano.NPeano.
Require Import Coq.Classes.SetoidClass.

Require Import fin.
Require Import pas.
Require Import vec.

Import FIN.NOTATIONS.
Import PAS.
Import PAS.NOTATIONS.
Import PAS.THMS.
Import VEC.NOTATIONS.

Local Open Scope FIN.
Local Open Scope VEC.
Local Open Scope PAS.

Module PCA.
```

## Definitions

```
  Class Pca `(pas : Pas) (k : carrier) (s : carrier) := {
    kSpec (a b : carrier) : &k*&a*&b ≃ &a;
    sSpec1 (a b : carrier) : &s*&a*&b!;
    sSpec2 (a b c: carrier) : &s*&a*&b*&c ≃ &a*&c*(&b*&c)
  }.

  Definition NonTrivial `(pca : Pca) : Prop := exists a b : carrier, ~ a == b.
```

### Identity combinator

```
  Definition identityCombinator `{_ : Pca} : term 0 := &s*&k*&k.
  Local Notation ι := identityCombinator.
```

### Term representations

```
  Definition representation' `{pca : Pca} {n : nat} (t : term (S n)) : term 0.
  Proof.
    induction n as [ | n IHn].
    - induction t as [a | x | u IHu v IHv].
      + exact (&k*&a).
      + exact ι.
      + exact (&s*IHu*IHv).
    - induction t as [a | x |u IHu v IHv].
      + exact (IHn (&k*&a)).
      + destruct x as [x | x].
        * exact (IHn (&k * var x)).
        * exact (IHn §ι).
      + set (i := idVec 0 (n := S n)).
        set (u' := §IHu ** i).
```

```
      set (v' := §IHv ** i).
      exact (IHn (&s*u'*v')).
Defined.
Local Notation λ' := representation'.

Definition representation `{_ : Pca} {n : nat} : term n -> term 0 :=
match n with
| O   => fun t => t
| S n => fun t => λ' t
end.
Local Notation λ := representation.
```

### κ combinator

```
Definition kAltCombinator `{_ : Pca} : term 0 := &k*ι.
Local Notation κ := kAltCombinator.
```

### Pairing combinator

```
Definition pairingCombinator `{_ : Pca} : term 0 := λ (x23*x03*x13).
Local Notation π := pairingCombinator.
```

### Projection combinators

```
Definition leftProjectionCombinator `{_ : Pca} : term 0 := λ (x01*&k).
Local Notation π1 := leftProjectionCombinator.

Definition rightProjectionCombinator `{_ : Pca} : term 0 := λ (x01 * §κ).
Local Notation π2 := rightProjectionCombinator.
```

### Combinators representing naturals

```
Fixpoint natRepresentationCombinator `{_ : Pca} (n : nat) : term 0 :=
match n with
| O   => ι
| S n => π * κ * (natRepresentationCombinator n)
end.
Local Notation "# n" := (natRepresentationCombinator n) (at level 5).
```

### Case combinator

```
Definition caseCombinator `{_ : Pca} : term 0 := λ (§π1 * x23 * x03 * x13).
Local Notation δ := caseCombinator.
```

### Successor combinator

```
Definition successorCombinator `{_ : Pca} : term 0 := π * κ.
Local Notation σ := successorCombinator.
```

### Predecessor combinator

```
  Definition predecessorCombinator `{_ : Pca} : term 0 :=
    λ (§π1 * x01 * §#0 * §π2 * x01).
  Local Notation ψ := predecessorCombinator.
```

### Fixed point combinator

```
  Definition fixedPointCombinator `{_ : Pca} : term 0 :=
    let u := λ (x12*(x02*x02*x12)) in u*u.
  Local Notation φ1 := fixedPointCombinator.
```

### Double fixed point combinator

```
  Definition doubleFixedPointCombinator `{_ : Pca} : term 0 :=
    let u := λ (x13*(x03*x03*x13)*x23) in u*u.
  Local Notation φ2 := doubleFixedPointCombinator.
```

### Primitive recursion combinator

```
(*
  Local Notation r' :=
    (λ (x25 * (§ψ * x35) * (x05 * x15 * x25 * (§ψ * x35) * &k))).
  Local Notation r :=
    (λ (§δ * (&k * x14) * (§r' * x04 * x14 * x24 * x34) * x34)).
*)
  Definition r' `{_ : Pca} :=
    (λ (x25 * (§ψ * x35) * (x05 * x15 * x25 * (§ψ * x35) * &k))).
  Definition r `{_ : Pca} :=
    (λ (§δ * (&k * x14) * (§r' * x04 * x14 * x24 * x34) * x34)).
  Definition primitiveRecursionCombinator `{_ : Pca} : term 0 :=
    λ (§φ2 * §r * x03 * x13 * x23 * &k).
  Local Notation ρ := primitiveRecursionCombinator.
```

### Addition combinator

```
  Definition additionCombinator `{_ : Pca} : term 0 :=
    λ (§ρ * x12 * (&k * §σ) * x02).
  Local Notation α := additionCombinator.
```

### Multiplication combinator

```
  Definition multiplicationCombinator `{_ : Pca} : term 0 :=
    λ (§ρ * §#0 * (&k * (§α * x12)) * x02).
  Local Notation μ := multiplicationCombinator.
```

## Notations

```
  Module NOTATIONS.

    Delimit Scope PCA with PCA.

    Notation ι := ι.
```

```
  Notation λ := λ.
  Notation κ := κ.
  Notation π := π.
  Notation π1 := π1.
  Notation π2 := π2.
  Notation δ := δ.
  Notation σ := σ.
  Notation ψ := ψ.
  Notation ψ1 := ψ1.
  Notation ψ2 := ψ2.
  Notation ρ := ρ.
  Notation α := α.
  Notation μ := μ.
  Notation "# n" := #n : PCA.

End NOTATIONS.
```

# Theorems

```
Module Type THMS_SIG.
```

## Immediate consequenses of axioms

```
  Axiom kSpec' : forall `{_ : Pca} (t : term 0) (b : carrier), &k*t*&b ≃ t.

  Axiom kaDenotes : forall `{_ : Pca} (a : carrier), &k*&a!.

  Axiom sSpec2' : forall `{_ : Pca} (u v w : term 0), &s*u*v*w ≃ u*w*(v*w).

  Axiom saDenotes : forall `{_ : Pca} (a : carrier), &s*&a!.
```

## Nontrivial if and only if k and s are distinct

```
  Axiom ksDistinctIffNonTrivial :
    forall `{pca : Pca}, ~ k == s <-> NonTrivial pca.
```

## Combinatory completeness

```
  Axiom combinatoryCompleteness1 :
    forall `{_ : Pca} {n : nat} (t : term (S n)) (f : VEC.t carrier n),
    λ t ** &&f !.

  Axiom combinatoryCompleteness2 :
    forall `{_ : Pca} {n : nat} (t : term n) (f : VEC.t carrier n),
    λ t ** &&f ≃ t/&&f.

  Axiom combinatoryCompleteness1' :
    forall `{_ : Pca} {n : nat} (t : term (S n)) (f : VEC.t (term 0) n),
    (forall x : FIN.t n, VEC.nth f x !) -> λ t ** f !.

  Axiom combinatoryCompleteness2' :
    forall `{_ : Pca} {n : nat} (t : term n) (f : VEC.t (term 0) n),
    (forall x : FIN.t n, VEC.nth f x !) -> λ t ** f ≃ t / f.
```

```
Axiom combinatoryCompletenessInstance1 :
  forall `{_ : Pca} (t : term 1) (a : carrier), λ t * &a ≃ t / &&((();; a)).

Axiom combinatoryCompletenessInstance2 :
  forall `{_ : Pca} (t : term 2) (a b : carrier),
  λ t * &a * &b ≃ t / &&((();; a;; b)).

Axiom combinatoryCompletenessInstance3 :
  forall `{_ : Pca} (t : term 3) (a b c : carrier),
  λ t * &a * &b * &c ≃ t / &&((();; a;; b;; c)).

Axiom combinatoryCompletenessInstance4 :
  forall `{_ : Pca} (t : term 4) (a b c d : carrier),
  λ t * &a * &b * &c * &d ≃ t / &&((();; a;; b;; c;; d)).

Axiom combinatoryCompletenessInstance5 :
  forall `{_ : Pca} (t : term 5) (a b c d e: carrier),
  λ t * &a * &b * &c * &d * &e ≃ t / &&((();; a;; b;; c;; d;; e)).
```

### Representations denote

```
Axiom representationsDenote :
  forall `{_ : Pca} {n : nat} (t : term (S n)), λ t !.

Axiom representationsDenote' :
  forall `{_ : Pca} {n m : nat} (t : term (S (m + n)))
         (f : VEC.t carrier n),
  λ t ** &&f!.

Axiom representationsDenote'' :
  forall `{_ : Pca} {n m : nat} (t : term (S n)) (f : VEC.t carrier m),
  Compare_dec.leb m n = true -> λ t ** &&f!.
```

### Properties and correctness of combinators

```
Axiom identityCombinatorCorrect :
  forall `{_ : Pca} (a : carrier), ι*&a ≃ &a.

Axiom identityCombinatorCorrect' : forall `{_ : Pca} (t : term 0), ι*t ≃ t.

Axiom identityCombinatorDenotes : forall `{_ : Pca}, ι!.

Axiom kAltCombinatorCorrect :
  forall `{_ : Pca} (a b : carrier), κ*&a*&b ≃ &b.

Axiom kAltCombinatorCorrect' :
  forall `{_ : Pca} (a : carrier) (t : term 0), κ*&a*t ≃ t.

Axiom kAltCombinatorDenotes : forall `{_ : Pca}, κ!.

Axiom kAltCombinatorDenotes' : forall `{_ : Pca} (a : carrier), κ*&a!.

Axiom pairingCombinatorDenotes : forall `{_ : Pca}, π!.
```

```
Axiom pairingCombinatorDenotes' : forall `{_ : Pca} (a : carrier), π*&a!.

Axiom pairingCombinatorDenotes'':
  forall `{_ : Pca} (a b : carrier), π*&a*&b!.

Axiom leftProjectionCombinatorCorrect :
  forall `{_ : Pca} (a b : carrier), π1*(π*&a*&b) ≃ &a.

Axiom leftProjectionCombinatorCorrect' :
  forall `{_ : Pca} (t : term 0) (b : carrier), π1*(π*t*&b) ≃ t.

Axiom leftProjectionCombinatorDenotes : forall `{_ : Pca}, π1!.

Axiom rightProjectionCombinatorCorrect :
  forall `{_ : Pca} (a b : carrier), π2*(π*&a*&b) ≃ &b.

Axiom rightProjectionCombinatorCorrect' :
  forall `{_ : Pca} (a : carrier) (t : term 0), π2*(π*&a*t) ≃ t.

Axiom rightProjectionCombinatorDenotes : forall `{_ : Pca}, π2!.

Axiom natRepresentationCombinatorDenotes : forall `{_ : Pca} (n : nat), #n!.

Axiom caseCombinatorCorrect0 :
  forall `{_ : Pca} (a b : carrier), δ*&a*&b*#0 ≃ &a.

Axiom caseCombinatorCorrect0' :
  forall `{_ : Pca} (t : term 0) (b : carrier), δ*t*&b*#0 ≃ t.

Axiom caseCombinatorCorrectS :
  forall `{_ : Pca} (n : nat) (a b : carrier), δ*&a*&b*#(S n) ≃ &b.

Axiom caseCombinatorCorrectS' :
  forall `{_ : Pca} (a : carrier) (t : term 0) (x : nat), δ*&a*t*#(S x) ≃ t.

Axiom caseCombinatorDenotes : forall `{_ : Pca}, δ!.

Axiom caseCombinatorDenotes' : forall `{_ : Pca} (a : carrier), δ*&a!.

Axiom caseCombinatorDenotes'' : forall `{_ : Pca} (a b : carrier), δ*&a*&b!.

Axiom successorCombinatorCorrect :
  forall `{_ : Pca} (n : nat), σ*#n ≃ #(S n).

Axiom successorCombinatorDenotes : forall `{_ : Pca}, σ!.

Axiom successorCombinatorDenotes' : forall `{_ : Pca} (n : nat), σ*#n!.

Axiom predecessorCombinatorCorrect0 : forall `{_ : Pca}, ψ*#0 ≃ #0.

Axiom predecessorCombinatorCorrectS :
  forall `{_ : Pca} (n : nat), ψ*#(S n) ≃ #n.

Axiom predecessorCombinatorCorrect :
```

```
  forall `{_ : Pca} (n : nat), ψ*#n ≃ #(n-1).

Axiom predecessorCombinatorDenotes :
  forall `{_ : Pca}, ψ!.

Axiom fixedPointCombinatorCorrect :
  forall `{_ : Pca} (a : carrier), ψ1*&a ≃ &a*(ψ1*&a).

Axiom fixedPointCombinatorCorrect' :
  forall `{_ : Pca} (t : term 0), ψ1*t ≃ t*(ψ1*t).

Axiom doubleFixedPointCombinatorCorrect :
  forall `{_ : Pca} (a b : carrier), ψ2*&a*&b ≃ &a*(ψ2*&a)*&b.

Axiom doubleFixedPointCombinatorDenotes : forall `{_ : Pca}, ψ2!.

Axiom doubleFixedPointCombinatorDenotes' :
  forall `{_ : Pca} (a : carrier), ψ2*&a!.

Axiom primitiveRecursionCombinatorCorrect0 :
  forall `{_ : Pca} (a b : carrier), ρ*&a*&b*#0 ≃ &a.

Axiom primitiveRecursionCombinatorCorrectS :
  forall `{_ : Pca} (a b : carrier) (x : nat),
  ρ*&a*&b*#(S x) ≃ &b*#x*(ρ*&a*&b*#x).

Axiom primitiveRecursionCombinatorDenotes : forall `{_ : Pca}, ρ!.

Axiom primitiveRecursionCombinatorDenotes' :
  forall `{_ : Pca} (a : carrier), ρ*&a!.

Axiom primitiveRecursionCombinatorDenotes'' :
  forall `{_ : Pca} (a b : carrier), ρ*&a*&b!.

Axiom additionCombinatorCorrect :
  forall `{_ : Pca} (x y : nat), α*#x*#y ≃ #(x + y).

Axiom additionCombinatorDenotes : forall `{_ : Pca}, α!.

Axiom additionCombinatorDenotes' : forall `{_ : Pca} (x : nat), α*#x!.

Axiom multiplicationCombinatorCorrect :
  forall `{_ : Pca} (x y : nat), μ*#x*#y ≃ #(x * y).

Axiom multiplicationCombinatorDenotes' :
  forall `{_ : Pca} (x : nat), μ*#x!.

Axiom multiplicationCombinatorDenotes :
  forall `{_ : Pca}, μ!.
```

## Arithmetic correct in nontrivial PCAs

```
Axiom correctArithmetic :
  forall `{pca : Pca} (x y : nat), NonTrivial pca -> #x ≃ #y -> x = y.
```

```
End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Theorem kSpec' `{pca : Pca} (t : term 0) (b : carrier) : &k*t*&b ≃ t.
  Proof.
    apply PAS.THMS.correctness2. intro a. split; intro H.
    - assert (&k*t*&b!) as H' by eauto.
      assert (t!) as [a' Ht] by eauto using subtermDenotes.
      rewrite <- H, Ht. symmetry. apply kSpec.
    - rewrite H. apply kSpec.
  Qed.

  Theorem kaDenotes `{pca : Pca} (a : carrier) : &k*&a!.
  Proof. cut (&k*&a*&a!); eauto using subtermDenotes, kSpec. Qed.

  Theorem sSpec2' `{pca : Pca} (u v w : term 0) : &s*u*v*w ≃ u*w*(v*w).
  Proof.
    apply correctness2. intro d. split; intro H; rewrite <- H.
    - assert (&s*u*v*w!) as H' by eauto.
      assert (u!) as [a Hua] by eauto using subtermDenotes.
      assert (v!) as [b Hvb] by eauto using subtermDenotes.
      assert (w!) as [c Hwc] by eauto using subtermDenotes.
      rewrite Hua, Hvb, Hwc. symmetry. apply sSpec2.
    - assert (u*w*(v*w)!) as H' by eauto.
      assert (u!) as [a Hua] by eauto using subtermDenotes.
      assert (v!) as [b Hvb] by eauto using subtermDenotes.
      assert (w!) as [c Hwc] by eauto using subtermDenotes.
      rewrite Hua, Hvb, Hwc. apply sSpec2.
  Qed.

  Theorem saDenotes `{pca : Pca} (a : carrier) : &s*&a!.
  Proof. cut (&s*&a*&a!); eauto using subtermDenotes, sSpec1. Qed.

  Theorem identityCombinatorCorrect `{pca : Pca} (a : carrier) : ι*&a ≃ &a.
  Proof.
    unfold ι.
    rewrite sSpec2.
    destruct (kaDenotes a) as [ka Hka]. rewrite Hka at 2.
    apply kSpec.
  Qed.

  Theorem identityCombinatorCorrect' `{pca : Pca} (t : term 0) : ι*t ≃ t.
  Proof.
    apply correctness2. intro a. split; intro H.
    - assert (t!) as [b Htb] by eauto using subtermDenotes. rewrite Htb in *.
      rewrite identityCombinatorCorrect in H. exact H.
    -  rewrite H. apply identityCombinatorCorrect.
  Qed.

  Theorem identityCombinatorDenotes `{pca : Pca} : ι!.
  Proof. apply sSpec1. Qed.
```

```
Theorem ksDistinctIffNonTrivial `{pca : Pca} : ~ k == s <-> NonTrivial pca.
Proof.
  unfold NonTrivial. split; [eauto | ].
  intros [a [b Hab]] Hks. apply Hab, constInjective.
  assert (forall a : carrier, &a ≃ &k) as H. {
    clear a b Hab. intro a.
    setoid_rewrite <- identityCombinatorCorrect at 1.
    unfold identityCombinator.
    rewrite <- Hks, kSpec at 1.
    setoid_rewrite <- identityCombinatorCorrect at 1.
    unfold identityCombinator.
    rewrite <- Hks, kSpec, kSpec.
    reflexivity.
  }
  setoid_rewrite H. reflexivity.
Qed.

Lemma representationDefinition' `{pca : Pca} {n : nat} (t : term (S n)) :
  representation' (n := n) t =
  (fun (n : nat) => match n with
  | O   => fun t => match t with
           | &a    => &k*&a
           | var _ => ι
           | u*v   => &s * λ' u * λ' v
           end
  | S n => fun t => match t with
           | &a    => λ' (n := n) (&k*&a)
           | var x => match x with
                      | inl x => λ' (n := n) (&k * var x)
                      | inr _ => λ' (n := n) §ι
                      end
           | u*v   =>
               let id := idVec 0 (n := S n) in
               λ' (&s * §(λ' u) ** id * §(λ' v) ** id)
           end
  end) n t.
Proof. destruct n; destruct t; auto. Qed.

Opaque representation'.

Theorem combinatoryCompleteness' `{pca : Pca} {n : nat} (t : term (S n))
                                 (f : VEC.t (term 0) (S n)) :
  (forall x : FIN.t (S n), VEC.nth f x !) -> λ' t ** f ≃ t / f.
Proof.
  revert t f.
  induction n as [ | n IHn].
  - intros t [[] t'] Hf. destruct (Hf FIN.last) as [a Ha]. simpl in Ha.
    unfold product. simpl.
    induction t as [b | x | u IHu v IHv]; rewrite representationDefinition'.
    + simpl. rewrite Ha. apply kSpec.
    + rewrite identityCombinatorCorrect'.
      setoid_rewrite FIN.THMS.t1Singleton.
      reflexivity.
    + rewrite sSpec2', IHu, IHv. reflexivity.
  - intros t [f t'] Hf.
```

```
    destruct (Hf FIN.last) as [b Hb]. simpl in Hb.
    assert (λ' t ** (f;; t') = λ' t ** f * t') as H by reflexivity.
    rewrite H. clear H.
    assert (forall x, VEC.nth f x!) as H. { intro x. exact (Hf ++x). }
    induction t as [a' | x | u IHu v IHv];
    rewrite representationDefinition'.
    + rewrite IHn; [ | exact H]. simpl. rewrite Hb. apply kSpec.
    + destruct x as [x | x].
      * rewrite IHn; [ | exact H]. simpl. rewrite Hb. apply kSpec'.
      * rewrite IHn; [ | exact H]. apply identityCombinatorCorrect'.
    + rewrite IHn; [ | exact H].
      simpl.
      setoid_rewrite (idVecProductSubstitutionEq0' _ f).
      setoid_rewrite sSpec2'.
      rewrite IHu, IHv.
      reflexivity.
Qed.

Theorem combinatoryCompleteness1' `{pca : Pca} {n : nat} (t : term (S n))
                                  (f : VEC.t (term 0) n) :
  (forall x : FIN.t n, VEC.nth f x !) -> λ t ** f !.
Proof.
  simpl. intro H.
  induction n.
  - unfold product. simpl.
    induction t as [a | x | u IHu v IHv]; rewrite representationDefinition'.
    + apply kaDenotes.
    + apply identityCombinatorDenotes.
    + destruct IHu as [a IHu], IHv as [b IHv].
      rewrite IHu, IHv. apply sSpec1.
  - induction t as [a | x | u [u' IHu] v [v' IHv]];
    rewrite representationDefinition'.
    + rewrite (combinatoryCompleteness' _ _ H). simpl. apply kaDenotes.
    + destruct x as [x | x].
      * rewrite (combinatoryCompleteness' _ _ H).
        unfold substitution, pasVecToTermVec.
        destruct (H x) as [a Ha]. rewrite Ha. apply kaDenotes.
      * rewrite
          (combinatoryCompleteness' _ _ H), closedTermSubstitutionEq0.
        apply identityCombinatorDenotes.
    + rewrite (combinatoryCompleteness' _ _ H).
      assert
        ((&s * §(λ' u) ** idVec 0 * §(λ' v) ** idVec 0) / f =
         (&s / f) * ((§(λ' u) ** idVec 0) / f) * ((§(λ' v) ** idVec 0) / f))
      as H' by reflexivity.
      setoid_rewrite H'. clear H'.
      setoid_rewrite idVecProductSubstitutionEq0'.
      rewrite IHu, IHv. simpl.
      apply sSpec1.
Qed.

Theorem combinatoryCompleteness2' `{pca : Pca} {n : nat} (t : term n)
                                  (f : VEC.t (term 0) n) :
  (forall x : FIN.t n, VEC.nth f x !) -> λ t ** f ≃ t / f.
Proof.
```

```
  intro H.
  destruct n as [ | n].
  - destruct f. simpl.
    rewrite emptySubstitutionEq, closedTermToClosedTermEq.
    reflexivity.
  - apply combinatoryCompleteness'. exact H.
Qed.

Theorem combinatoryCompleteness1 `{pca : Pca} {n : nat} (t : term (S n))
                                 (f : VEC.t carrier n) :
  λ t ** &&f !.
Proof.
  apply combinatoryCompleteness1'. intro x.
  unfold pasVecToTermVec. rewrite VEC.THMS.nthMapEq.
  exists (VEC.nth f x). reflexivity.
Qed.

Theorem combinatoryCompleteness2 `{pca : Pca} {n : nat} (t : term n)
                                 (f : VEC.t carrier n) :
  λ t ** &&f ≃ t/&&f.
Proof.
  apply combinatoryCompleteness2'. intro x.
  unfold pasVecToTermVec. rewrite VEC.THMS.nthMapEq.
  exists (VEC.nth f x). reflexivity.
Qed.

Opaque representation.

Theorem representationsDenote `{pca : Pca} {n : nat} (t : term (S n)) :
  λ t !.
Proof.
  set (f := VEC.copies n k).
  apply (subtermDenotesProduct _ &&f).
  apply combinatoryCompleteness1.
Qed.

Theorem representationsDenote' `{pca : Pca} {n m : nat}
                               (t : term (S (m + n)))
                               (f : VEC.t carrier n) :
  λ t ** &&f!.
Proof.
  set (g := VEC.copies m k).
  apply (subtermDenotesProduct _ &&g).
  rewrite doubleProductEq.
  unfold pasVecToTermVec. rewrite <- VEC.THMS.mapConcatEq.
  apply combinatoryCompleteness1.
Qed.

Theorem representationsDenote'' `{pca : Pca} {n m : nat} (t : term (S n))
                                (f : VEC.t carrier m) :
  Compare_dec.leb m n = true -> λ t ** &&f!.
Proof.
  rewrite Compare_dec.leb_iff.
  intro H. apply NPeano.Nat.le_exists_sub in H. destruct H as [k' [H _]].
  subst. apply representationsDenote'.
```

```
Qed.

Theorem combinatoryCompletenessInstance1 `{pca : Pca} (t : term 1)
                                         (a : carrier) :
  λ t * &a ≃ t / &&((();; a)).
Proof.
  cutrewrite (λ t * &a = λ t ** &&((();; a)); [ | reflexivity].
  apply combinatoryCompleteness2.
Qed.

Theorem combinatoryCompletenessInstance2 `{pca : Pca} (t : term 2)
                                         (a b : carrier) :
  λ t * &a * &b ≃ t / &&((();; a;; b)).
Proof.
  cutrewrite (λ t * &a*&b = λ t ** &&((();; a;; b)); [ | reflexivity].
  apply combinatoryCompleteness2.
Qed.

Theorem combinatoryCompletenessInstance3 `{pca : Pca} (t : term 3)
                                         (a b c : carrier) :
  λ t * &a * &b * &c ≃ t / &&((();; a;; b;; c)).
Proof.
  cutrewrite (λ t * &a*&b*&c = λ t ** &&((();; a;; b;; c)); [ | reflexivity].
  apply combinatoryCompleteness2.
Qed.

Theorem combinatoryCompletenessInstance4 `{pca : Pca} (t : term 4)
                                         (a b c d : carrier) :
  λ t * &a * &b * &c * &d ≃ t / &&((();; a;; b;; c;; d)).
Proof.
  cutrewrite (λ t * &a*&b*&c*&d = λ t ** &&((();; a;; b;; c;; d));
    [ | reflexivity].
  apply combinatoryCompleteness2.
Qed.

Theorem combinatoryCompletenessInstance5 `{pca : Pca} (t : term 5)
                                         (a b c d e: carrier) :
  λ t * &a * &b * &c * &d * &e ≃ t / &&((();; a;; b;; c;; d;; e)).
Proof.
  cutrewrite (λ t * &a*&b*&c*&d*&e = λ t ** &&((();; a;; b;; c;; d;; e));
    [ | reflexivity].
  apply combinatoryCompleteness2.
Qed.

Theorem kAltCombinatorCorrect `{pca : Pca} (a b : carrier) : κ*&a*&b ≃ &b.
Proof. unfold κ. rewrite kSpec'. apply identityCombinatorCorrect. Qed.

Theorem kAltCombinatorCorrect' `{pca : Pca} (a : carrier) (t : term 0) :
  κ*&a*t ≃ t.
Proof. unfold κ. rewrite kSpec'. apply identityCombinatorCorrect'. Qed.

Theorem kAltCombinatorDenotes `{pca : Pca} : κ!.
Proof.
  unfold κ. destruct identityCombinatorDenotes as [i Hi]. rewrite Hi.
  apply kaDenotes.
```

```
Qed.

Theorem kAltCombinatorDenotes' `{pca : Pca} (a : carrier) : κ*&a!.
Proof.
  cut (κ*&a*&a!); eauto using subtermDenotes, kAltCombinatorCorrect.
Qed.

Theorem pairingCombinatorDenotes `{pca : Pca} : π!.
Proof. apply representationsDenote. Qed.

Theorem pairingCombinatorDenotes'' `{pca : Pca} (a b : carrier) : π*&a*&b!.
Proof.
  cutrewrite (π*&a*&b = π**&&((), a, b) : VEC.t carrier 2));
  [ | reflexivity].
  apply combinatoryCompleteness1.
Qed.

Theorem pairingCombinatorDenotes' `{pca : Pca} (a : carrier) : π*&a!.
Proof.
  cut (π*&a*&a!); eauto using subtermDenotes, pairingCombinatorDenotes''.
Qed.

Theorem leftProjectionCombinatorCorrect `{pca : Pca} (a b : carrier) :
  π1*(π*&a*&b) ≃ &a.
Proof.
  destruct (pairingCombinatorDenotes'' a b) as [p Hp]. rewrite Hp.
  unfold π1. rewrite combinatoryCompletenessInstance1. simpl.
  rewrite <- Hp. unfold π. setoid_rewrite combinatoryCompletenessInstance3.
  apply kSpec.
Qed.

Theorem leftProjectionCombinatorCorrect' `{pca : Pca} (t : term 0)
                                         (b : carrier) :
  π1*(π*t*&b) ≃ t.
Proof.
  apply correctness2. intro a. split; intro H.
  - assert (t!) as [a' Ht] by eauto 7 using subtermDenotes. rewrite Ht in *.
    rewrite leftProjectionCombinatorCorrect in H. exact H.
  - rewrite H. apply leftProjectionCombinatorCorrect.
Qed.

Theorem leftProjectionCombinatorDenotes `{pca : Pca} : π1!.
Proof.
  cut (π1*(π*&k*&k)!);
  eauto using subtermDenotes, leftProjectionCombinatorCorrect.
Qed.

Theorem rightProjectionCombinatorCorrect `{pca : Pca} (a b : carrier) :
  π2*(π*&a*&b) ≃ &b.
Proof.
  destruct (pairingCombinatorDenotes'' a b) as [p Hp]. rewrite Hp.
  unfold π2. rewrite combinatoryCompletenessInstance1. simpl.
  destruct (kAltCombinatorDenotes) as [k' Hk'].
  rewrite Hk', <- Hp. unfold π. rewrite combinatoryCompletenessInstance3.
  simpl. rewrite <- Hk'. apply kAltCombinatorCorrect.
```

```
Qed.

Theorem rightProjectionCombinatorCorrect' `{pca : Pca} (a : carrier)
                                          (t : term 0) :
  π2*(π*&a*t) ≃ t.
Proof.
  apply correctness2. intro b. split; intro H.
  - assert (t!) as [b' Ht] by eauto 6 using subtermDenotes. rewrite Ht in *.
    rewrite rightProjectionCombinatorCorrect in H. exact H.
  - rewrite H. apply rightProjectionCombinatorCorrect.
Qed.

Theorem rightProjectionCombinatorDenotes `{pca : Pca} : π2!.
Proof.
  assert (π2*(π*&k*&k)!) by eauto using rightProjectionCombinatorCorrect.
  eauto using subtermDenotes, rightProjectionCombinatorCorrect.
Qed.

Theorem natRepresentationCombinatorDenotes `{pca : Pca} (n : nat) : #n!.
Proof.
  induction n as [ | n [a IHn]].
  - apply identityCombinatorDenotes.
  - unfold natRepresentationCombinator. rewrite IHn.
    destruct kAltCombinatorDenotes as [k' Hk']. rewrite Hk'.
    apply pairingCombinatorDenotes''.
Qed.

Theorem caseCombinatorCorrect0 `{pca : Pca} (a b : carrier) :
  δ*&a*&b*#0 ≃ &a.
Proof.
  simpl. destruct identityCombinatorDenotes as [i Hi]. rewrite Hi.
  setoid_rewrite combinatoryCompletenessInstance3. simpl.
  setoid_rewrite closedTermSubstitutionEq0.
  setoid_rewrite combinatoryCompletenessInstance1. simpl.
  rewrite <- Hi, identityCombinatorCorrect'.
  apply kSpec.
Qed.

Theorem caseCombinatorCorrect0' `{pca : Pca} (t : term 0) (b : carrier) :
  δ*t*&b*#0 ≃ t.
Proof.
  apply correctness2. intro a. split; intro H.
  - assert (t!) as [a' Ht] by eauto 7 using subtermDenotes. rewrite Ht in *.
    rewrite caseCombinatorCorrect0 in H. exact H.
  - rewrite H. apply caseCombinatorCorrect0.
Qed.

Theorem caseCombinatorCorrectS `{pca : Pca} (n : nat) (a b : carrier) :
  δ*&a*&b*#(S n) ≃ &b.
Proof.
  destruct
    (natRepresentationCombinatorDenotes n) as [n' Hn],
    (natRepresentationCombinatorDenotes (S n)) as [Sn HSn].
  unfold δ. rewrite HSn, combinatoryCompletenessInstance3. simpl.
  setoid_rewrite closedTermSubstitutionEq0.
```

```
  rewrite <- HSn. simpl. rewrite Hn, leftProjectionCombinatorCorrect'.
  apply kAltCombinatorCorrect.
Qed.

Theorem caseCombinatorCorrectS' `{pca : Pca} (a : carrier) (t : term 0)
                                (x : nat) :
  δ*&a*t*#(S x) ≃ t.
Proof.
  apply correctness2. intro b. split; intro H.
  - assert (t!) as [b' Ht] by eauto 7 using subtermDenotes. rewrite Ht in *.
    rewrite caseCombinatorCorrectS in H. exact H.
  - rewrite H. apply caseCombinatorCorrectS.
Qed.

Theorem caseCombinatorDenotes'' `{pca : Pca} (a b : carrier) : δ*&a*&b!.
Proof.
  assert (δ*&a*&b*#0!) by eauto using caseCombinatorCorrect0.
  destruct (natRepresentationCombinatorDenotes 0) as [x Hx].
  rewrite Hx in H. eauto using subtermDenotes.
Qed.

Theorem caseCombinatorDenotes' `{pca : Pca} (a : carrier) : δ*&a!.
Proof.
  pose proof (caseCombinatorDenotes'' a a). eauto using subtermDenotes.
Qed.

Theorem caseCombinatorDenotes `{pca : Pca} : δ!.
Proof.
  pose proof (caseCombinatorDenotes' k). eauto using subtermDenotes.
Qed.

Theorem successorCombinatorCorrect `{pca : Pca} (n : nat) : σ*#n ≃ #(S n).
Proof. reflexivity. Qed.

Theorem successorCombinatorDenotes' `{pca : Pca} (n : nat) : σ*#n!.
Proof.
  rewrite successorCombinatorCorrect.
  apply natRepresentationCombinatorDenotes.
Qed.

Theorem successorCombinatorDenotes `{pca : Pca} : σ!.
Proof.
  pose proof (successorCombinatorDenotes' 0). eauto using subtermDenotes.
Qed.

Theorem predecessorCombinatorCorrect0 `{pca : Pca} : ψ*#0 ≃ #0.
Proof.
  unfold natRepresentationCombinator.
  destruct
    identityCombinatorDenotes as [i Hi],
    rightProjectionCombinatorDenotes as [p2 Hp2].
  rewrite Hi.
  unfold ψ. rewrite combinatoryCompletenessInstance1. simpl.
  setoid_rewrite closedTermSubstitutionEq0.
  assert (π1*&i ≃ &k) as H. {
```

```
    unfold π1. rewrite combinatoryCompletenessInstance1. simpl.
    rewrite <- Hi. apply identityCombinatorCorrect.
  }
  rewrite H, Hp2, Hi, kSpec, <- Hi.
  apply identityCombinatorCorrect'.
Qed.

Theorem predecessorCombinatorCorrectS `{pca : Pca} (n : nat) :
  ψ*#(S n) ≃ #n.
Proof.
  destruct
    (natRepresentationCombinatorDenotes n) as [n' Hn],
    (natRepresentationCombinatorDenotes (S n)) as [Sn HSn].
  rewrite Hn, HSn.
  unfold ψ. rewrite combinatoryCompletenessInstance1. simpl.
  rewrite closedTermSubstitutionEq0, closedTermSubstitutionEq0.
  rewrite <- HSn. simpl. rewrite Hn at 1.
  rewrite leftProjectionCombinatorCorrect'.
  destruct
    identityCombinatorDenotes as [i Hi],
    kAltCombinatorDenotes as [k' Hk'].
  rewrite
    Hi, kAltCombinatorCorrect', Hk', rightProjectionCombinatorCorrect', Hn.
  reflexivity.
Qed.

Theorem predecessorCombinatorCorrect `{pca : Pca} (n : nat) : ψ*#n ≃ #(n-1).
Proof.
  destruct n as [ | n].
  - rewrite predecessorCombinatorCorrect0. reflexivity.
  - rewrite predecessorCombinatorCorrectS. simpl.
    rewrite <- Minus.minus_n_O.
    reflexivity.
Qed.

Theorem predecessorCombinatorDenotes `{pca : Pca} : ψ!.
Proof. apply representationsDenote. Qed.

Theorem fixedPointCombinatorCorrect `{pca : Pca} (a : carrier) :
  ψ1*&a ≃ &a*(ψ1*&a).
Proof.
  unfold ψ1 at 1. simpl.
  set (u := x12*(x02*x02*x12)). unfold u at 1.
  destruct (representationsDenote u) as [u' Hu].
  rewrite Hu, combinatoryCompletenessInstance2. simpl. rewrite <- Hu.
  reflexivity.
Qed.

Theorem fixedPointCombinatorCorrect' `{pca : Pca} (t : term 0) :
  ψ1*t ≃ t*(ψ1*t).
Proof.
  apply correctness2. intro a. split; intro H.
  - assert (t!) as [b Ht] by eauto using subtermDenotes.
    rewrite Ht, <- H in *. symmetry.
    apply fixedPointCombinatorCorrect.
```

```
  - assert (t!) as [b Ht] by eauto using subtermDenotes.
    rewrite Ht, <- H in *.
    apply fixedPointCombinatorCorrect.
Qed.

Theorem doubleFixedPointCombinatorDenotes `{pca : Pca} : ψ2!.
Proof.
  unfold ψ2.
  set (u := x13*(x03*x03*x13)*x23). unfold u at 1.
  destruct (representationsDenote u) as [u' Hu]. rewrite Hu. clear Hu.
  fold u. cutrewrite (λ u * &u' = λ u ** &&((();; u'));  [ | reflexivity].
  apply representationsDenote''. reflexivity.
Qed.

Theorem doubleFixedPointCombinatorDenotes' `{pca : Pca} (a : carrier) :
  ψ2*&a!.
Proof.
  unfold ψ2.
  set (u := x13*(x03*x03*x13)*x23). unfold u at 1.
  destruct (representationsDenote u) as [u' Hu].
  rewrite Hu. fold u.
  cutrewrite (λ u * &u' * &a = λ u ** &&((();; u';; a));  [ | reflexivity].
  apply combinatoryCompleteness1.
Qed.

Theorem doubleFixedPointCombinatorCorrect `{pca : Pca} (a b : carrier) :
  ψ2*&a*&b ≃ &a*(ψ2*&a)*&b.
Proof.
  unfold ψ2 at 1. simpl.
  set (u := x13*(x03*x03*x13)*x23). unfold u at 1.
  destruct (representationsDenote u) as [u' Hu].
  rewrite Hu, combinatoryCompletenessInstance3. simpl. rewrite <- Hu.
  reflexivity.
Qed.

Lemma primitiveRecursionCombinatorEq `{pca : Pca} (a b : carrier)
                                     (x : nat) :
  ρ*&a*&b*#x ≃ δ*(&k*&a)*(r'*(ψ2*r)*&a*&b*#x)*#x*&k.
Proof.
  destruct (natRepresentationCombinatorDenotes x) as [x' Hx]. rewrite Hx.
  unfold ρ. rewrite combinatoryCompletenessInstance3. simpl.
  setoid_rewrite closedTermSubstitutionEq0.
  assert (r!) as Hr by eauto using representationsDenote.
  destruct Hr as [r'' Hr]. rewrite Hr.
  setoid_rewrite doubleFixedPointCombinatorCorrect.
  destruct (doubleFixedPointCombinatorDenotes' r'') as [ψr Hψr].
  setoid_rewrite Hψr.
  setoid_rewrite <- Hr.
  setoid_rewrite combinatoryCompletenessInstance4. simpl.
  rewrite closedTermSubstitutionEq0.
  rewrite closedTermSubstitutionEq0.
  rewrite <- Hψr, <- Hr.
  reflexivity.
Qed.
```

```
Lemma rDenotes' `{pca : Pca} (a b : carrier) (x : nat) :
  r'*(ψ2*r)*&a*&b*#x!.
Proof.
  assert (r!) as [c Hc] by eauto using representationsDenote.
  destruct
    (doubleFixedPointCombinatorDenotes' c) as [d Hd],
    (natRepresentationCombinatorDenotes x) as [e He].
  rewrite Hc, Hd, He.
  cutrewrite (r' * &d * &a * &b * &e = r' ** &&((();; d;; a;; b;; e));
    [ | reflexivity].
  apply combinatoryCompleteness1.
Qed.

Theorem primitiveRecursionCombinatorCorrect0 `{pca : Pca} (a b : carrier) :
  ρ*&a*&b*#0 ≃ &a.
Proof.
  rewrite primitiveRecursionCombinatorEq.
  destruct (rDenotes' a b 0) as [r'' Hr].
  rewrite Hr, caseCombinatorCorrect0', kSpec.
  reflexivity.
Qed.

Theorem primitiveRecursionCombinatorCorrectS `{pca : Pca} (a b : carrier)
                                             (x : nat) :
  ρ*&a*&b*#(S x) ≃ &b*#x*(ρ*&a*&b*#x).
Proof.
  rewrite primitiveRecursionCombinatorEq.
  destruct (rDenotes' a b (S x)) as [r'' Hr''].
  destruct (kaDenotes a) as [ka Hka].
  rewrite Hr'', Hka, caseCombinatorCorrectS', <- Hr''.
  assert (r!) as [r''' Hr'''] by eauto using representationsDenote.
  destruct
    (natRepresentationCombinatorDenotes x) as [x' Hx],
    (natRepresentationCombinatorDenotes (S x)) as [Sx HSx],
    (doubleFixedPointCombinatorDenotes' r''') as [ψr Hψr].
  unfold r, r' in *.
  rewrite Hx, HSx, Hr''', Hψr, combinatoryCompletenessInstance5. simpl.
  setoid_rewrite closedTermSubstitutionEq0.
  rewrite <- HSx, predecessorCombinatorCorrect, <- Hψr, <- Hr'''. simpl.
  rewrite <- Minus.minus_n_O.
  unfold ρ. rewrite combinatoryCompletenessInstance3. simpl.
  rewrite closedTermSubstitutionEq0.
  rewrite closedTermSubstitutionEq0.
  rewrite <- Hx.
  reflexivity.
Qed.

Theorem primitiveRecursionCombinatorDenotes `{pca : Pca} : ρ!.
Proof. apply representationsDenote. Qed.

Theorem primitiveRecursionCombinatorDenotes'' `{pca : Pca} (a b : carrier) :
  ρ*&a*&b!.
Proof.
  cutrewrite (ρ*&a*&b = ρ ** &&((();; a;; b)); [ | reflexivity].
  apply combinatoryCompleteness1.
```

```
Qed.

Theorem primitiveRecursionCombinatorDenotes' `{pca : Pca} (a : carrier) :
  ρ*&a!.
Proof.
  pose proof (primitiveRecursionCombinatorDenotes'' a a) as H.
  eauto using subtermDenotes.
Qed.

Theorem additionCombinatorCorrect `{pca : Pca} (x y : nat) :
  α*#x*#y ≃ #(x + y).
Proof.
  unfold α.
  destruct
    (natRepresentationCombinatorDenotes x) as [x' Hx],
    (natRepresentationCombinatorDenotes y) as [y' Hy],
    successorCombinatorDenotes as [S HS].
  destruct (kaDenotes S) as [kS HkS].
  rewrite Hx, Hy, combinatoryCompletenessInstance2. simpl.
  rewrite closedTermSubstitutionEq0.
  rewrite closedTermSubstitutionEq0.
  rewrite <- Hx, HS, HkS. clear x' Hx.
  induction x as [ | x IHx].
  - setoid_rewrite primitiveRecursionCombinatorCorrect0. rewrite <- Hy.
    reflexivity.
  - setoid_rewrite primitiveRecursionCombinatorCorrectS.
    destruct (natRepresentationCombinatorDenotes x) as [x' Hx].
    rewrite IHx, Hx, <- HkS, <- HS, kSpec', successorCombinatorCorrect.
    reflexivity.
Qed.

Theorem additionCombinatorDenotes' `{pca : Pca} (x : nat) : α*#x!.
Proof.
  assert (α*#x*#x!) as H. {
    destruct (natRepresentationCombinatorDenotes (x + x)) as [xx Hxx].
    exists xx. rewrite additionCombinatorCorrect, <- Hxx. reflexivity.
  }
  eauto using subtermDenotes.
Qed.

Theorem additionCombinatorDenotes `{pca : Pca} : α!.
Proof.
  pose proof (additionCombinatorDenotes' 0) as H.
  eauto using subtermDenotes.
Qed.

Theorem multiplicationCombinatorCorrect `{pca : Pca} (x y : nat) :
  μ*#x*#y ≃ #(x * y).
Proof.
  unfold μ.
  destruct
    (natRepresentationCombinatorDenotes x) as [x' Hx],
    (natRepresentationCombinatorDenotes y) as [y' Hy],
    (additionCombinatorDenotes' y) as [αy Hαy].
  rewrite Hx, Hy, combinatoryCompletenessInstance2. simpl.
```

```
  rewrite closedTermSubstitutionEq0.
  rewrite closedTermSubstitutionEq0.
  destruct
    (kaDenotes αy) as [kαy Hkαy], identityCombinatorDenotes as [i Hi].
  rewrite <- Hx, <- Hy, Hαy, Hkαy, Hi. clear x' Hx.
  induction x as [ | x IHx].
  - rewrite primitiveRecursionCombinatorCorrect0, Hi. reflexivity.
  - rewrite primitiveRecursionCombinatorCorrectS.
    destruct (natRepresentationCombinatorDenotes x) as [x' Hx].
    rewrite IHx, Hx, <- Hkαy, <- Hαy, kSpec', additionCombinatorCorrect.
    reflexivity.
Qed.

Theorem multiplicationCombinatorDenotes' `{pca : Pca} (x : nat) : μ*#x!.
Proof.
  assert (μ*#x*#x!) as H. {
    destruct (natRepresentationCombinatorDenotes (x * x)) as [xx Hxx].
    exists xx. rewrite multiplicationCombinatorCorrect, <- Hxx. reflexivity.
  }
  eauto using subtermDenotes.
Qed.

Theorem multiplicationCombinatorDenotes `{pca : Pca} : μ!.
Proof.
  pose proof (multiplicationCombinatorDenotes' 0) as H.
  eauto using subtermDenotes.
Qed.

Theorem correctArithmetic `{pca : Pca} (x y : nat) :
  NonTrivial pca -> #x ≠ #y -> x = y.
Proof.
  rewrite <- ksDistinctIffNonTrivial. intro ksDistinct.
  assert (forall x, ~ #(S x) ≈ #0) as H. {
    clear x y. simpl. intros x H. apply ksDistinct.
    destruct (natRepresentationCombinatorDenotes x) as [x' Hx'].
    rewrite Hx' in H.
    pose proof (leftProjectionCombinatorCorrect' κ x') as H'.
    rewrite H in H'.
    unfold π1 in H'.
    destruct identityCombinatorDenotes as [i Hi]. rewrite Hi in H'.
    rewrite combinatoryCompletenessInstance1 in H'. simpl in H'.
    rewrite <- Hi, identityCombinatorCorrect in H'.
    apply constInjective.
    pose proof (kSpec k s) as H''.
    rewrite H' in H'' at 1.
    rewrite kAltCombinatorCorrect in H''.
    symmetry in H''.
    exact H''.
  }
  clear ksDistinct.
  revert y. induction x as [ | x IHx].
  - intro y. destruct y as [ | y]; [tauto | ].
    intro H'. symmetry in H'. apply H in H'. contradiction.
  - destruct y as [ | y].
    + intro H'. apply H in H'. contradiction.
```

```
        + simpl. intro H'.
          destruct kAltCombinatorDenotes as [k' Hk']. rewrite Hk' in *.
          pose proof (rightProjectionCombinatorCorrect' k' #x) as Hx.
          pose proof (rightProjectionCombinatorCorrect' k' #y) as Hy.
          rewrite <- H', Hx in Hy. rewrite (IHx _ Hy). reflexivity.
    Qed.

  End THMS.
```

Hide implementations.

```
  Global Opaque representation.
  Global Opaque ι λ κ π π1 π2 δ σ ψ ψ1 ψ2 ρ α μ natRepresentationCombinator.

End PCA.
```

# I. pcarealizability.v—Realizability with partial combinatory algebras

```
Require Import Coq.Classes.Morphisms.

Require Import fin.
Require Import heytingarithmetic.
Require Import pas.
Require Import pca.
Require Import standardha.
Require Import vec.

Import FIN.NOTATIONS.
Import HA.NOTATIONS.
Import PAS.NOTATIONS.
Import PCA.NOTATIONS.
Import VEC.NOTATIONS.

Local Open Scope FIN.
Local Open Scope VEC.
Local Open Scope HA.
Local Open Scope PAS.
Local Open Scope PCA.

Local Notation x01 := PAS.x01.
Local Notation x02 := PAS.x02.
Local Notation x12 := PAS.x12.
Local Notation x03 := PAS.x03.
Local Notation x13 := PAS.x13.
Local Notation x23 := PAS.x23.

Module PCA_REL.
```

## Definitions

### Vector of naturals to vector of representations

```
Local Notation "## f" :=
  (VEC.map (fun x => #x) f) (at level 6, right associativity).
```

### Realizability

```
Definition RealizabilityPred `{_ : PCA.Pca} (n : nat) :=
  VEC.t nat n -> PAS.term 0 -> Prop.

Definition AtomicRealizability `{_ : PCA.Pca} {n} (P : HA.atom n)
  : RealizabilityPred n :=
  fun f t => STD_HA.TruthPred P f /\ t!.

Fixpoint Realizability `{_ : PCA.Pca} {n} (P : HA.formula n)
  : RealizabilityPred n :=
match P with
| HA.fAtom P => AtomicRealizability P
| P ∧ Q      =>
    fun f t => Realizability P f (π1*t) /\ Realizability Q f (π2*t)
| P ∨ Q      =>
    fun f t => (π1*t ≃ #0 /\ Realizability P f (π2*t)) \/
               (π1*t ≃ #1 /\ Realizability Q f (π2*t))
| P → Q      =>
    fun f u => u! /\ forall v, Realizability P f v -> Realizability Q f (u*v)
| ∃P         =>
    fun f t => exists n : nat, π1*t ≃ #n /\ Realizability P (f, n) (π2*t)
| ∀P         => fun f t => forall x : nat, Realizability P (f, x) (t*#x)
end.

Definition IsRealizable `{_ : PCA.Pca} {n} (P : HA.formula n) : Prop :=
  exists t : PAS.term 0, forall f : VEC.t nat n, Realizability P f (t**##f).
```

## Notations

```
Module NOTATIONS.

  Delimit Scope PCA_REL with PCA_REL.

  Notation "## f" := ##f (at level 6, right associativity) : PCA_REL.

End NOTATIONS.
```

## Theorems

```
Module Type THMS_SIG.

  Declare Instance realizabilityInterpretation `{_ : PCA.Pca} :
    HA.Interpretation (@IsRealizable _ _ _ _ _ _ _).

  Declare Instance realizabilityRespectsClosedTermEq `{_ : PCA.Pca} {n}
                                                     (A : HA.formula n)
```

```
                                                    (f : VEC.t nat n) :
    Proper (PAS.ClosedTermEq ==> iff) (Realizability A f).

  Axiom realizersDenote :
    forall `{_ : PCA.Pca} {n} {A : HA.formula n} {f : VEC.t nat n}
           {t : PAS.term 0},
      Realizability A f t -> t!.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Lemma realizabilityRespectful `{pca : PCA.Pca} {n} (A : HA.formula n)
                                (f : VEC.t nat n) (u v : PAS.term 0) :
    u ≃ v -> Realizability A f u -> Realizability A f v.
  Proof.
    generalize dependent u.
    generalize dependent v.
    induction A as
      [n A | n A IHA B IHB | n A IHA B IHB | n A IHA B IHB | n A IH | n A IH];
    intros u v Huv; simpl.
    - unfold AtomicRealizability. rewrite Huv. tauto.
    - intros [HA HB]. split.
      + apply (IHA f (π1*u) (π1*v)).
        * rewrite Huv. reflexivity.
        * exact HA.
      + apply (IHB f (π2*u) (π2*v)).
        * rewrite Huv. reflexivity.
        * exact HB.
    - intros [[H H'] | [H H']].
      + left. split.
        * rewrite <- Huv. exact H.
        * apply (IHA f (π2*u) (π2*v)); [ | exact H']. rewrite Huv.
          reflexivity.
      + right. split.
        * rewrite <- Huv. exact H.
        * apply (IHB f (π2*u) (π2*v)); [ | exact H']. rewrite Huv.
          reflexivity.
    - intros [Hv H]. split; [rewrite <- Huv; exact Hv | ].
      intros w Hw. specialize (H w Hw). apply IHB with (u := v*w).
      + rewrite Huv. reflexivity.
      + exact H.
    - intros [x [Hx H]]. exists x. split; [rewrite <- Huv; exact Hx | ].
      apply IH with (u := π2*v).
      + rewrite Huv. reflexivity.
      + exact H.
    - intros H x. apply IH with (u := v*#x).
      + rewrite Huv. reflexivity.
      + apply H.
  Qed.

  Instance realizabilityRespectsClosedTermEq `{pca : PCA.Pca} {n}
                                             (A : HA.formula n)
```

```
                                          (f : VEC.t nat n) :
  Proper (PAS.ClosedTermEq ==> iff) (Realizability A f).
Proof.
  intros u v Huv.
  split; intro H; [ | symmetry in Huv]; eauto using realizabilityRespectful.
Qed.

Theorem realizersDenote `{pca : PCA.Pca} {n} {A : HA.formula n}
                        {f : VEC.t nat n} {t : PAS.term 0} :
  Realizability A f t -> t!.
Proof.
  generalize dependent t.
  induction A as
    [n A | n A IHA B _ | n A IHA B IHB | n A _ B _ | n A IH | n A IH];
  simpl; intro t.
  - unfold AtomicRealizability. tauto.
  - intros [H _]. eauto using PAS.THMS.subtermDenotes.
  - intros [ [H H'] | [H H'] ]; eauto using PAS.THMS.subtermDenotes.
  - tauto.
  - intros [x [H H'] ]. eauto using PAS.THMS.subtermDenotes.
  - intros H. specialize (H 0). eauto using PAS.THMS.subtermDenotes.
Qed.

Lemma realizabilitySubstitution `{pca : PCA.Pca} {n m} (A : HA.formula n)
                                (f : VEC.t (HA.term m) n) (g : VEC.t nat m)
                                (t : PAS.term 0) :
  Realizability (A // f) g t <-> Realizability  A (STD_HA.termVecVal f g) t.
Proof.
  generalize dependent m. generalize dependent t.
  induction A as
    [n A | n A IHA B IHB | n A IHA B IHB | n A IHA B IHB | n A IH | n A IH];
  intros m f g; simpl.
  - unfold AtomicRealizability.
    setoid_rewrite STD_HA.THMS.atomicTruthSubstEq.
    tauto.
  - setoid_rewrite IHA. setoid_rewrite IHB. tauto.
  - setoid_rewrite IHA. setoid_rewrite IHB. tauto.
  - setoid_rewrite IHA. setoid_rewrite IHB. tauto.
  - setoid_rewrite IH. setoid_rewrite STD_HA.THMS.termVecValTermVecUpEq'.
    tauto.
  - setoid_rewrite IH. setoid_rewrite STD_HA.THMS.termVecValTermVecUpEq'.
    tauto.
Qed.

Lemma realizabilityLastSubstitution `{pca : PCA.Pca} {n}
                                    (A : HA.formula (S n))
                                    (t : HA.term (S n)) (f : VEC.t nat n)
                                    (x : nat) (u : PAS.term 0) :
  Realizability (A /+ t) (f, x) u <->
  Realizability A (f, STD_HA.termVal t (f, x)) u.
Proof.
  setoid_rewrite realizabilitySubstitution.
  assert
    (STD_HA.termVecVal (HA.idVecUp;; t) (f;; x) =
     (STD_HA.termVecVal HA.idVecUp (f;; x));; STD_HA.termVal t (f;; x))
```

```
  as H by reflexivity.
  rewrite H. clear H.
  setoid_rewrite STD_HA.THMS.termVecValIdVecUpEq.
  tauto.
Qed.

Lemma realizabilityDownSubstitution `{pca : PCA.Pca} {n}
                                    (A : HA.formula (S n)) (t : HA.term n)
                                    (f : VEC.t nat n) (u : PAS.term 0) :
  Realizability (A /- t) f u <-> Realizability A (f, STD_HA.termVal t f) u.
Proof.
  setoid_rewrite realizabilitySubstitution.
  assert
    (STD_HA.termVecVal (HA.idVec;; t) f =
     (STD_HA.termVecVal HA.idVec f);; STD_HA.termVal t f)
  as H by reflexivity.
  rewrite H. clear H.
  setoid_rewrite STD_HA.THMS.termVecValIdVecEq.
  tauto.
Qed.

Lemma realizabilityUp `{pca : PCA.Pca} {n} (A : HA.formula n)
                      (f : VEC.t nat n) (x : nat) (t : PAS.term 0) :
  Realizability (up A) (f, x) t <-> Realizability A f t.
Proof.
  unfold HA.formulaUp.
  rewrite realizabilitySubstitution, STD_HA.THMS.termVecValIdVecUpEq.
  tauto.
Qed.

Lemma combinatoryCompleteness1 `{pca : PCA.Pca} {n} (t : PAS.term (S n))
                               (f : VEC.t nat n) :
  λ t ** ##f!.
Proof.
  apply PCA.THMS.combinatoryCompleteness1'.
  intro x. rewrite VEC.THMS.nthMapEq.
  apply PCA.THMS.natRepresentationCombinatorDenotes.
Qed.

Lemma combinatoryCompleteness2 `{pca : PCA.Pca} {n} (t : PAS.term n)
                               (f : VEC.t nat n) :
  λ t ** ##f ≃ t/##f.
Proof.
  apply PCA.THMS.combinatoryCompleteness2'.
  intro x. rewrite VEC.THMS.nthMapEq.
  apply PCA.THMS.natRepresentationCombinatorDenotes.
Qed.

Lemma combinatoryCompleteness2' `{pca : PCA.Pca} {n} (t : PAS.term (S n))
                                (f : VEC.t nat n) (a : carrier) :
  λ t ** ##f * &a ≃ t/(##f, &a).
Proof.
  cutrewrite (λ t ** ##f * &a = λ t ** (##f;; &a)); [ | reflexivity].
  apply PCA.THMS.combinatoryCompleteness2'.
  destruct x as [x | x]; simpl.
```

```
  - rewrite VEC.THMS.nthMapEq.
    apply PCA.THMS.natRepresentationCombinatorDenotes.
  - exists a. reflexivity.
Qed.

Fixpoint termRep `{pca : PCA.Pca} {n} (t : HA.term n) : PAS.term 0 :=
match t with
| HA.O'       => λ (n := n) §#0
| HA.var x    => λ (PAS.var x)
| HA.S' t     => λ (n := n) (§σ * §(termRep t) ** PAS.idVec 0)
| u + v       =>
  let u' := §(termRep u) ** PAS.idVec 0 in
  let v' := §(termRep v) ** PAS.idVec 0 in
  λ (n := n) (§α * u' * v')
| u · v       =>
  let u' := §(termRep u) ** PAS.idVec 0 in
  let v' := §(termRep v) ** PAS.idVec 0 in
  λ (n := n) (§μ * u' * v')
end.

Lemma termRepCorrect `{pca : PCA.Pca} {n} (t : HA.term n)
                     (f : VEC.t nat n) :
  (termRep t)**##f ≃ #(STD_HA.termVal t f).
Proof.
  induction t as [ | x | t IHt | u IHu v IHv | u IHu v IHv];
  simpl; rewrite combinatoryCompleteness2; simpl.
  - setoid_rewrite PAS.THMS.closedTermSubstitutionEq0. reflexivity.
  - rewrite VEC.THMS.nthMapEq. reflexivity.
  - setoid_rewrite PAS.THMS.idVecProductSubstitutionEq0'.
    setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
    rewrite IHt.
    rewrite PCA.THMS.successorCombinatorCorrect.
    reflexivity.
  - setoid_rewrite PAS.THMS.idVecProductSubstitutionEq0'.
    setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
    rewrite IHu, IHv.
    rewrite PCA.THMS.additionCombinatorCorrect.
    reflexivity.
  - setoid_rewrite PAS.THMS.idVecProductSubstitutionEq0'.
    setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
    rewrite IHu, IHv.
    rewrite PCA.THMS.multiplicationCombinatorCorrect.
    reflexivity.
Qed.

Instance realizabilityInterpretation `{pca : PCA.Pca} :
  HA.Interpretation (@IsRealizable _ _ _ _ _ _ _).
Proof.
  split.
  - intros [t H]. specialize (H ()). destruct H as [H _]. exact H.
  - intros n A B [u Hu] [v Hv].
    exists (λ (n := n) (§u ** PAS.idVec 0 * §v ** PAS.idVec 0)).
    intro f.
    specialize (Hu f). specialize (Hv f).
    rewrite combinatoryCompleteness2.
```

```
  simpl.
  setoid_rewrite PAS.THMS.idVecProductSubstitutionEq0'.
  destruct Hu as [_ Hu]. exact (Hu _ Hv).
- intros n A [t H]. exists t. intro f. simpl. intro x. apply H.
- intros n A [t H].
  exists (λ (n := (S n)) (§t ** PAS.idVec 1)).
  intros [f x].
  rewrite combinatoryCompleteness2, realizabilityUp.
  assert (##(f;; x) = (###f;; #x)) as H' by reflexivity.
  rewrite H'. clear H'.
  setoid_rewrite PAS.THMS.idVecProductSubstitutionEq1'.
  apply H.
- intros n A B.
  exists (λ (n := n) &k).
  intro f.
  rewrite combinatoryCompleteness2. simpl.
  split; [exists k; reflexivity | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hua].
  setoid_rewrite Hua. rewrite Hua in Ha. clear dependent u.
  split; [apply PCA.THMS.kaDenotes | ].
  intros v Hb. destruct (realizersDenote Hb) as [b Hvb].
  rewrite Hvb in *. clear dependent v.
  setoid_rewrite PCA.kSpec. exact Ha.
- intros n A B C.
  exists (λ (n := n) &s).
  intro f. rewrite combinatoryCompleteness2.
  split; [exists s; reflexivity | ].
  intros t Ha. destruct (realizersDenote Ha) as [a Ht]. rewrite Ht in *.
  clear dependent t.
  split; [ apply PCA.THMS.saDenotes | ].
  intros t Hb. destruct (realizersDenote Hb) as [b Ht]. rewrite Ht in *.
  clear dependent t.
  split; [ apply PCA.sSpec1 | ].
  intros t Hc. destruct (realizersDenote Hc) as [c Ht]. rewrite Ht in *.
  clear dependent t.
  setoid_rewrite PCA.THMS.sSpec2'.
  destruct Ha as [_ Ha], Hb as [_ Hb].
  specialize (Ha _ Hc). specialize (Hb _ Hc).
  destruct Ha as [_ Ha]. exact (Ha _ Hb).
- intros n A B. exists (λ (n := n) §π). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [ apply PCA.THMS.pairingCombinatorDenotes | ].
  intros t Ha. destruct (realizersDenote Ha) as [a Ht]. rewrite Ht in *.
  clear dependent t.
  split; [apply PCA.THMS.pairingCombinatorDenotes' | ].
  intros t Hb. destruct (realizersDenote Hb) as [b Ht]. rewrite Ht in *.
  clear dependent t.
  split.
  + rewrite PCA.THMS.leftProjectionCombinatorCorrect. exact Ha.
  + rewrite PCA.THMS.rightProjectionCombinatorCorrect. exact Hb.
- intros n A B. exists (λ (n := n) §π1). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [apply PCA.THMS.leftProjectionCombinatorDenotes | ].
```

```
  simpl. tauto.
- intros n A B. exists (λ (n := n) §π2). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [apply PCA.THMS.rightProjectionCombinatorDenotes | ].
  simpl. tauto.
- intros n A B. exists (λ (n := n) (§(π * #0))). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  destruct (PCA.THMS.natRepresentationCombinatorDenotes 0) as [x Hx].
  rewrite Hx.
  split; [apply PCA.THMS.pairingCombinatorDenotes' | ].
  intros t Ha. destruct (realizersDenote Ha) as [a Ht]. rewrite Ht in *.
  clear dependent t.
  left. split.
  + rewrite PCA.THMS.leftProjectionCombinatorCorrect, Hx. reflexivity.
  + rewrite PCA.THMS.rightProjectionCombinatorCorrect. exact Ha.
- intros n A B. exists (λ (n := n) (§(π * #1))). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  destruct (PCA.THMS.natRepresentationCombinatorDenotes 1) as [x Hx].
  rewrite Hx.
  split; [apply PCA.THMS.pairingCombinatorDenotes' | ].
  intros t Ha. destruct (realizersDenote Ha) as [a Ht]. rewrite Ht in *.
  clear dependent t.
  right. split.
  + rewrite PCA.THMS.leftProjectionCombinatorCorrect, Hx. reflexivity.
  + rewrite PCA.THMS.rightProjectionCombinatorCorrect. exact Ha.
- intros n A B C.
  set (t := §δ * x13 * x23 * (§π1 * x03) * (§π2 * x03)). simpl in t.
  exists (λ (n := n) §(λ t)). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [apply PCA.THMS.representationsDenote | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hu]. rewrite Hu in *.
  clear dependent u.
  split. {
    apply PAS.THMS.subtermDenotes with (v := λ t * &a * &a); [auto | ].
    cutrewrite (λ t * &a * &a = λ t ** &&((();; a;; a)); [ | reflexivity].
    apply PCA.THMS.combinatoryCompleteness1.
  }
  intros u Hb. destruct (realizersDenote Hb) as [b Hu]. rewrite Hu in *.
  clear dependent u.
  split. {
    cutrewrite (λ t * &a * &b = λ t ** &&((();; a;; b)); [ | reflexivity].
    apply PCA.THMS.combinatoryCompleteness1.
  }
  intros u Hc. destruct (realizersDenote Hc) as [c Hu]. rewrite Hu in *.
  clear dependent u.
  rewrite PCA.THMS.combinatoryCompletenessInstance3. simpl.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  destruct Ha as [[H H'] | [H H']]; rewrite H.
  + setoid_rewrite PCA.THMS.caseCombinatorCorrect0.
    destruct Hb as [_ Hb]. exact (Hb _ H').
  + setoid_rewrite PCA.THMS.caseCombinatorCorrectS.
```

```
    destruct Hc as [_ Hc]. exact (Hc _ H').
- intros n A. exists (λ (n := n) &k). intro f.
  rewrite combinatoryCompleteness2. simpl.
  split; [exists k; reflexivity | ]. intros v [H _]. contradiction.
- intros n A t.
  exists (λ (n := S n) (§π * §(termRep t) ** PAS.idVec 1 * PAS.lastVar)).
  intro f.
  split; [apply combinatoryCompleteness1 | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hu]. rewrite Hu in *.
  clear dependent u.
  rewrite realizabilityDownSubstitution in Ha.
  set (x := STD_HA.termVal t f). exists x.
  rewrite combinatoryCompleteness2'. simpl.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  setoid_rewrite PAS.THMS.idVecProductSubstitutionEq1'.
  rewrite termRepCorrect. fold x. fold x in Ha.
  destruct (PCA.THMS.natRepresentationCombinatorDenotes x) as [x' Hx].
  rewrite Hx.
  split.
  + apply PCA.THMS.leftProjectionCombinatorCorrect.
  + setoid_rewrite PCA.THMS.rightProjectionCombinatorCorrect. exact Ha.
- intros n A B.
  set (t := λ (x02 * (§π1 * x12) * (§π2 * x12))).
  exists (λ (n := n) §t). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [ apply PCA.THMS.representationsDenote | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hu]. rewrite Hu in *.
  clear dependent u.
  split. {
    cutrewrite (t * &a  = t ** &&((();; a)); [ | reflexivity].
    apply PCA.THMS.combinatoryCompleteness1.
  }
  intros u Hb. destruct (realizersDenote Hb) as [b Hu]. rewrite Hu in *.
  clear dependent u.
  unfold t. rewrite PCA.THMS.combinatoryCompletenessInstance2. simpl.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  destruct Hb as [x [Hx Hb]]. rewrite Hx in *.
  simpl in Ha. specialize (Ha x). destruct Ha as [_ Ha].
  specialize (Ha _ Hb). apply realizabilityUp in Ha. exact Ha.
- intros n A B.
  set (t := λ (x03*x23*x13)).
  exists (λ (n := n) §t). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [apply PCA.THMS.representationsDenote | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hu]. rewrite Hu in *.
  clear dependent u.
  split. {
    apply PAS.THMS.subtermDenotes with (v := t*&a*&a); [auto | ].
    cutrewrite (t * &a * &a  = t ** &&((();; a;; a)); [ | reflexivity].
    apply PCA.THMS.combinatoryCompleteness1.
  }
  intros u Hb. destruct (realizersDenote Hb) as [b Hu]. rewrite Hu in *.
  clear dependent u.
```

```
  simpl. intro x.
  destruct (PCA.THMS.natRepresentationCombinatorDenotes x) as [x' Hx].
  rewrite Hx in *.
  unfold t. rewrite PCA.THMS.combinatoryCompletenessInstance3. simpl.
  simpl in Ha. specialize (Ha x). destruct Ha as [_ Ha].
  setoid_rewrite realizabilityUp in Ha.
  rewrite <- Hx. exact (Ha _ Hb).
- intros n A t.
  exists (λ (n := S n) (PAS.lastVar * §(termRep t) ** PAS.idVec 1)).
  split; [apply combinatoryCompleteness1 | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hu]. rewrite Hu in *.
  clear dependent u.
  rewrite realizabilityDownSubstitution, combinatoryCompleteness2'. simpl.
  setoid_rewrite PAS.THMS.idVecProductSubstitutionEq1'.
  rewrite termRepCorrect.
  apply Ha.
- exists (λ (n := 1) &k).
  intros [f x]. destruct f.
  rewrite combinatoryCompleteness2. simpl.
  split; [ | exists k; reflexivity].
  simpl. reflexivity.
- exists (λ (n := 2) &k).
  intros [[f x] y]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha [a Hua]]. rewrite Hua. clear dependent u.
  simpl. split; [ | apply PCA.THMS.kaDenotes].
  simpl in *. congruence.
- exists (λ (n := 3) &k).
  intros [[[f x] y] z]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha [a Hua]]. rewrite Hua. clear dependent u.
  split; [apply PCA.THMS.kaDenotes | ].
  intros u [Hb [b Hub]]. rewrite Hub. clear dependent u.
  simpl. split; [ | eauto using PCA.kSpec].
  simpl in *. congruence.
- exists (λ (n := 2) &k).
  intros [[f x] y]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha [a Hua]]. rewrite Hua. clear dependent u.
  simpl. split; [ | apply PCA.THMS.kaDenotes].
  simpl in *. congruence.
- exists (λ (n := 4) &k).
  intros [[[[f x1] x2] x3] x4]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha [a Hua]]. rewrite Hua. clear dependent u.
  split; [apply PCA.THMS.kaDenotes | ].
  intros u [Hb [b Hub]]. rewrite Hub. clear dependent u.
  simpl. split; [ | eauto using PCA.kSpec].
  simpl in *. congruence.
- exists (λ (n := 4) &k).
  intros [[[[f x1] x2] x3] x4]. destruct f.
```

```
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha [a Hua]]. rewrite Hua. clear dependent u.
  split; [apply PCA.THMS.kaDenotes | ].
  intros u [Hb [b Hub]]. rewrite Hub. clear dependent u.
  simpl. split; [ | eauto using PCA.kSpec].
  simpl in *. congruence.
- exists (λ (n := 1) &k).
  intros [f x]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha _]. simpl in Ha. exfalso. eapply n_Sn. eauto.
- exists (λ (n := 2) &k).
  intros [[f x] y]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [exists k; reflexivity | ].
  intros u [Ha [a Hua]]. rewrite Hua. clear dependent u.
  simpl. split; [ | apply PCA.THMS.kaDenotes].
  simpl in *. congruence.
- exists (λ (n := 1) &k).
  intros [f x]. destruct f.
  rewrite combinatoryCompleteness2. simpl.
  split; [ | exists k; reflexivity].
  simpl. auto.
- exists (λ (n := 2) &k).
  intros [[f x] y]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [ | exists k; reflexivity].
  simpl. auto.
- exists (λ (n := 1) &k).
  intros [f x]. destruct f.
  rewrite combinatoryCompleteness2. simpl.
  split; [ | exists k; reflexivity].
  simpl. auto.
- exists (λ (n := 2) &k).
  intros [[f x] y]. destruct f.
  rewrite combinatoryCompleteness2.
  split; [ | exists k; reflexivity].
  simpl. auto.
- intros n A.
  exists (λ (n := n) §ρ). intro f.
  rewrite combinatoryCompleteness2.
  setoid_rewrite PAS.THMS.closedTermSubstitutionEq0.
  split; [apply PCA.THMS.primitiveRecursionCombinatorDenotes | ].
  intros u Ha. destruct (realizersDenote Ha) as [a Hu]. rewrite Hu in *.
  clear dependent u.
  split; [apply PCA.THMS.primitiveRecursionCombinatorDenotes' | ].
  intros u Hb. destruct (realizersDenote Hb) as [b Hu]. rewrite Hu in *.
  clear dependent u.
  simpl. intro x.
  induction x as [ | x IHx].
  + setoid_rewrite PCA.THMS.primitiveRecursionCombinatorCorrect0.
    rewrite realizabilityDownSubstitution in Ha. exact Ha.
  + setoid_rewrite PCA.THMS.primitiveRecursionCombinatorCorrectS.
    destruct (realizersDenote IHx) as [c Hc]. rewrite Hc in *.
```

```
          simpl in Hb. specialize (Hb x). destruct Hb as [_ Hb].
          specialize (Hb _ IHx).
          setoid_rewrite realizabilityLastSubstitution in Hb. exact Hb.
    Qed.

  End THMS.

End PCA_REL.
```

# J. rewriting.v—Some results relating to rewriting systems

The proof that the diamond property for the reflexive closure implies closure follows "Polishing up the Tait–Martin-Lof proof of the Church–Rosser theorem" (Robert Pollack 1995).

```
Require Import Coq.Relations.Operators_Properties.
Require Import Coq.Relations.Relation_Operators.
Require Import Coq.Setoids.Setoid.

Module RW.
```

## Definitions

### Reflexive closure

```
  Definition ReflexiveClosure {A} (R : A -> A -> Prop) : A -> A -> Prop :=
    fun a b => R a b \/ a = b.
  Local Notation RC := ReflexiveClosure.
```

### Reflexive transitive closure

```
  Local Notation RTC := (clos_refl_trans _).
```

### Generalized diamond property

```
  Definition GeneralizedDiamondProperty {A} (R S : A -> A -> Prop) : Prop :=
    forall a b c : A, R a b -> S a c -> exists d : A, S b d /\ R c d.
  Local Notation GDP := GeneralizedDiamondProperty.
```

### Diamond property

```
  Definition DiamondProperty {A} (R : A -> A -> Prop) : Prop := GDP R R.
  Local Notation DP := DiamondProperty.
```

### Confluence

```
  Definition Confluence {A} (R : A -> A -> Prop) := DP (RTC R).
```

### Functional relation

```
Definition Functional {A} (R : A -> A -> Prop) : Prop :=
  forall a b c : A, R a b -> R a c -> b = c.
```

### Normal elements and normal forms

```
Definition Normal {A} (R : A -> A -> Prop) (a : A) : Prop :=
  ~ exists b : A, R a b.

Definition NormalForm {A} (R : A -> A -> Prop) (a b : A) : Prop :=
  RTC R a b /\ Normal R b.
Local Notation NF := NormalForm.
```

## Notations

```
Module NOTATIONS.

  Notation RC := RC.
  Notation RTC := RTC.
  Notation GDP := GDP.
  Notation DP := DP.
  Notation NF := NF.

End NOTATIONS.
```

## Theorems

```
Module Type THMS_SIG.

  Axiom rtcExtensive :
    forall {A} (R : A -> A -> Prop) (a b : A), R a b -> RTC R a b.

  Axiom rtcIncreasing :
    forall {A} (R S : A -> A -> Prop),
    (forall a b : A, R a b -> S a b) ->
    (forall a b : A, RTC R a b -> RTC S a b).
```

### Diamond property implies confluence

```
  Axiom dp_confluence : forall {A} (R : A -> A -> Prop), DP R -> Confluence R.
```

### Diamond property of reflexive closure implies confluence

```
  Axiom dpReflClos_confluence :
    forall {A} (R : A -> A -> Prop), DP (RC R) -> Confluence R.
```

### Functional relations are confluent

```
  Axiom functionalRelation_confluence :
    forall {A} (R : A -> A -> Prop), Functional R -> Confluence R.
```

### Normal reduction is trivial

```
  Axiom normalReductionTrivial :
    forall {A} (R : A -> A -> Prop) (a b : A),
    Normal R a -> RTC R a b -> a = b.
```

### Normal forms are unique for confluent relations

```
  Axiom confluence_normalFormsUnique :
    forall {A} (R : A -> A -> Prop),
    Confluence R -> forall a b c : A, NF R a b -> NF R a c -> b = c.

  Axiom functionalDoubleReduction :
    forall {A} (R : A -> A -> Prop),
    Functional R ->
    forall a b c : A, RTC R a b -> RTC R a c -> RTC R b c \/ RTC R c b.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Theorem rtcExtensive {A} (R : A -> A -> Prop) (a b : A) :
    R a b -> RTC R a b.
  Proof. auto using rt_step. Qed.

  Theorem rtcIncreasing {A} (R S : A -> A -> Prop) :
    (forall a b : A, R a b -> S a b) ->
    (forall a b : A, RTC R a b -> RTC S a b).
  Proof.
    intros H a b H'.
    induction H';
    [auto using rt_step | apply clos_rt_is_preorder | eauto using rt_trans].
  Qed.

  Lemma gdpCommutative {A} (R S : A -> A -> Prop) : GDP R S -> GDP S R.
  Proof. intros H a b c HR HS. specialize (H a c b HS HR). firstorder. Qed.

  Definition StripProperty {A} (f : (A -> A -> Prop) -> (A -> A -> Prop)) :=
    forall R S : (A -> A -> Prop), GDP R S -> GDP R (f S).
  Local Notation Strip := StripProperty.

  Definition StripsProperty {A} (f : (A -> A -> Prop) -> (A -> A -> Prop)) :=
    forall R S : (A -> A -> Prop), GDP R S -> GDP (f R) (f S).
  Local Notation Strips := StripsProperty.

  Lemma strip_strips {A} (f : (A -> A -> Prop) -> (A -> A -> Prop)) :
    Strip f -> Strips f.
  Proof. unfold Strips. auto using gdpCommutative. Qed.

  Lemma rtcStrip {A} : Strip (clos_refl_trans A).
  Proof.
    intros R S H.
    apply gdpCommutative.
```

```
  apply gdpCommutative in H.
  intros a b c H'. generalize dependent c.
  induction H' as [a b H' | a | a b' b HL IHL HR IHR].
  - intros c H''.
    destruct (H a b c H' H'') as [d [H''' H4]]. eauto using rt_step.
  - eauto using rt_refl.
  - intros c H'.
    destruct (IHL c H') as [b'' [H'' H''']].
    destruct (IHR b'' H'') as [d [H4 H5]]. eauto using rt_trans.
Qed.

Theorem dp_confluence {A} (R : A -> A -> Prop) : DP R -> Confluence R.
Proof. apply strip_strips. apply rtcStrip. Qed.

Theorem dpReflClos_confluence {A} (R : A -> A -> Prop) :
  DP (RC R) -> Confluence R.
Proof.
  set (R' := RC R).
  assert (forall u v, (RTC R') u v <-> (RTC R) u v) as H. {
    intros u v. split; intro H.
    - induction H.
      + inversion_clear H.
        * apply rt_step; assumption.
        * subst. apply rt_refl.
      + apply rt_refl.
      + eauto using rt_trans.
    - induction H.
      + apply rt_step. left. assumption.
      + apply rt_refl.
      + eauto using rt_trans.
  }
  intros H' u v v' H'' H'''. apply H in H''. apply H in H'''.
  destruct (dp_confluence _  H' u v v' H'' H''') as [w [Hvw Hv'w]].
  apply H in Hvw. apply H in Hv'w. eauto.
Qed.

Theorem functionalDoubleReduction {A} (R : A -> A -> Prop) :
  Functional R ->
  forall a b c : A, RTC R a b -> RTC R a c -> RTC R b c \/ RTC R c b.
Proof.
  setoid_rewrite clos_rt_rt1n_iff.
  intros HR a b c Hab Hac.
  induction Hab as [a | a a' b Haa' Ha'b IH]; [tauto | ].
  destruct Hac as [ | a'' c Haa'' Ha''c].
  - right. apply rt1n_trans with (y := a').
    + exact Haa'.
    + exact Ha'b.
  - specialize (HR _ _ _ Haa' Haa''). subst. apply IH. exact Ha''c.
Qed.

Theorem functionalRelation_confluence {A} (R : A -> A -> Prop) :
  Functional R -> Confluence R.
Proof.
  intros H a b c Hab Hac.
  destruct (functionalDoubleReduction _ H _ _ _ Hab Hac); firstorder.
```

```
    Qed.

    Theorem normalReductionTrivial {A} (R : A -> A -> Prop) (a b : A) :
      Normal R a -> RTC R a b -> a = b.
    Proof.
      intros H H'.
      induction H' as [U V H' | U H' | U V W _ IHL _ IHR].
      - exfalso. eauto.
      - reflexivity.
      - specialize (IHL H). subst. exact (IHR H).
    Qed.

    Theorem confluence_normalFormsUnique {A} (R : A -> A -> Prop) :
      Confluence R -> forall a b c : A, NF R a b -> NF R a c -> b = c.
    Proof.
      intros H a b c [Hab Hb] [Hac Hc].
      destruct (H _ _ _ Hab Hac) as [d [Hbd Hcd]].
      apply (normalReductionTrivial _ _ _ Hb) in Hbd.
      apply (normalReductionTrivial _ _ _ Hc) in Hcd.
      congruence.
    Qed.

  End THMS.

End RW.
```

# K. ksrewriting.v—Rewriting with K and S

```
Local Ltac inj_subst H := injection H; intros; subst; clear H.

Require Import Coq.Classes.Morphisms.
Require Import Coq.Relations.Operators_Properties.
Require Import Coq.Relations.Relation_Operators.
Require Import Coq.Setoids.Setoid.

Require Import pas.
Require Import pca.
Require Import rewriting.

Import PAS.NOTATIONS.
Import RW.NOTATIONS.

Local Open Scope PAS.

Module KS_RW.
```

## Definitions

### Terms

```
  Inductive term :=
  | K    : term
  | S    : term
```

```
| appl : term -> term -> term.
Local Infix "·" := appl (at level 40, left associativity).
```

### Terms to closed PCA terms

```
Fixpoint termToClosedPcaTerm `{_ : PCA.Pca} (T : term) : PAS.term 0 :=
match T with
| K   => &k
| S   => &s
| U·V => termToClosedPcaTerm U * termToClosedPcaTerm V
end.
Local Notation "$ t" := (termToClosedPcaTerm t) (at level 6).
```

### Non-deterministic reduction

```
Local Reserved Infix ">" (at level 70).
Inductive OneStepReduces : term -> term -> Prop :=
| oneStepReducesK U V    : K·U·V > U
| oneStepReducesS U V W  : S·U·V·W > U·W·(V·W)
| oneStepReducesL U U' V : U > U' -> U·V > U'·V
| oneStepReducesR U V V' : V > V' -> U·V > U·V'
where "U > V" := (OneStepReduces U V).

Definition Reduces := RTC OneStepReduces.
Local Infix ">*" := Reduces (at level 70).
```

### Convertibility

```
Definition Convertible (U V : term) : Prop := exists W, U >* W /\ V >* W.
Local Infix "≈" := Convertible (at level 70).
```

### Deterministic eager reduction

```
Fixpoint eagerReduct (T : term) : option term :=
match T with
| K
| S   => None
| U·V => match eagerReduct U, eagerReduct V with
         | Some U,      _ => Some (U·V)
         |   None, Some V => Some (U·V)
         |   None,   None => match T with
                             | K·U·V   => Some U
                             | S·U·V·W => Some (U·W·(V·W))
                             | _       => None
                             end
         end
end.

Definition EagerlyOneStepReduces (U V : term) : Prop :=
  eagerReduct U = Some V.
Local Infix ">>" := EagerlyOneStepReduces (at level 70).

Definition EagerlyReduces : term -> term -> Prop :=
  RTC (fun U V : term => U >> V).
```

```
Infix ">>*" := EagerlyReduces (at level 70).
```

### Normal terms and normal forms

```
Notation Normal := (RW.Normal OneStepReduces).

Local Notation NF := (NF OneStepReduces).

Definition StrictlyNormal (T : term) := eagerReduct T = None.
Local Notation SN := StrictlyNormal.

Definition StrictNormalForm (U V : term) : Prop := U >>* V /\ SN V.
Local Notation SNF := StrictNormalForm.
```

## Notations

```
Module NOTATIONS.

  Delimit Scope KS_RW with KS_RW.

  Notation "§ T" := (§T) (at level 6) : KS_RW.
  Infix "·" := appl : KS_RW.
  Infix ">" := (fun U V => U > V) (at level 70) : KS_RW.
  Infix ">*" := (fun U V => U >* V) (at level 70) : KS_RW.
  Infix ">>" := (fun U V => U >> V) (at level 70) : KS_RW.
  Infix ">>*" := (fun U V => U >>* V) (at level 70) : KS_RW.
  Infix "≈" := (fun U V => U ≈ V) (at level 70) : KS_RW.
  Notation Normal := Normal.
  Notation NF := NF.
  Notation SNF := SNF.

End NOTATIONS.
```

## Theorems

```
Module Type THMS_SIG.
```

### Convertibility is an equivalence relation

```
  Declare Instance convertibleEquivalence : Equivalence Convertible.
```

### Application respects convertibility

```
  Declare Instance applRespectsConvertibility :
    Proper (Convertible ==> Convertible ==> Convertible) appl.
```

### Normal forms respects convertibility

```
  Declare Instance normalFormsRespectConvertibility :
    Proper (Convertible ==> eq ==> iff) NF.
```

### Induction principle for eager reduction

```
Axiom eagerReductionInduction :
  forall P : term -> term -> Prop,
  (forall U U' V : term, P U U' -> P (U·V) (U'·V)) ->
  (forall U V V' : term, SN U -> P V V' -> P (U·V) (U·V')) ->
  (forall U V : term, SN U -> SN V -> P (K·U·V) U) ->
  (forall U V W : term, SN U -> SN V -> SN W -> P (S·U·V·W) (U·W·(V·W))) ->
  forall U V : term, U >> V -> P U V.
```

### Induction principle for normal terms

```
Axiom normalTermInduction :
  forall P : term -> Prop,
  P K ->
  P S ->
  (forall T : term, Normal T -> P T -> P (K·T)) ->
  (forall T : term, Normal T -> P T -> P (S·T)) ->
  (forall U V : term, Normal U -> Normal V -> P U -> P V -> P (S·U·V)) ->
  forall T : term, Normal T -> P T.

Axiom strictlyNormalTermInduction :
  forall P : term -> Prop,
  P K ->
  P S ->
  (forall T : term, SN T -> P T -> P (K·T)) ->
  (forall T : term, SN T -> P T -> P (S·T)) ->
  (forall U V : term, SN U -> SN V -> P U -> P V -> P (S·U·V)) ->
  forall T : term, SN T -> P T.
```

### Relations between reduction relations

```
Axiom eagerlyOneStepReduces_oneStepReduces :
  forall U V : term, U >> V -> U > V.

Axiom eagerlyReduces_reduces : forall U V : term, U >>* V -> U >* V.
```

### Properties of reduction

```
Axiom reductionL : forall U U' V : term, U >* U' -> U·V >* U'·V.

Axiom reductionR : forall U V V' : term,  V >* V' -> U·V >* U·V'.

Axiom reductionPar :
  forall U V U' V' : term,  U >* U' -> V >* V' -> U·V >* U'·V'.
```

### Properties of eager reduction

```
Axiom eagerOneStepReductionFunctional : RW.Functional EagerlyOneStepReduces.

Axiom eagerReductionL : forall U U' V : term, U >>* U' -> U·V >>* U'·V.

Axiom eagerReductionR :
  forall U V V' : term, SN U -> V >>* V' -> U·V >>* U·V'.
```

```
Axiom eagerReductionPar :
  forall U U' V V' : term, SN U' -> U >>* U' -> V >>* V' -> U·V >>* U'·V'.
```

### Reduction relations are confluent

```
Axiom reductionConfluent : RW.Confluence OneStepReduces.

Axiom eagerReductionConfluent : RW.Confluence EagerlyOneStepReduces.
```

### Definition of strict normal terms is correct

```
Axiom strictlyNormalCorrect :
  forall T : term, SN T <-> RW.Normal EagerlyOneStepReduces T.

Axiom strictNormalFormsCorrect :
  forall U V : term, SNF U V <-> RW.NormalForm EagerlyOneStepReduces U V.
```

### Properties of normal terms

```
Axiom normalIffStrictlyNormal : forall T, Normal T <-> SN T.

Axiom strictNormalForm_normalForm : forall U V : term, SNF U V -> NF U V.

Axiom normalApplL : forall U V : term, Normal (U·V) -> Normal U.

Axiom strictlyNormalApplL : forall U V : term, SN (U·V) -> SN U.

Axiom normalApplR : forall U V : term, Normal (U·V) -> Normal V.

Axiom strictlyNormalApplR : forall U V : term, SN (U·V) -> SN V.

Axiom normalReduction : forall U V : term, Normal U -> U >* V -> U = V.

Axiom strictlyNormalEagerReduction :
  forall U V : term, SN U -> U >>* V -> U = V.

Axiom strictNormalFormApplL :
  forall U V W : term, SNF (U·V) W -> exists U' : term, SNF U U'.

Axiom strictNormalFormApplR :
  forall U V W : term, SNF (U·V) W -> exists V' : term, SNF V V'.

Axiom normalFormsUnique :
  forall U V V' : term,  NF U V -> NF U V' -> V = V'.

Axiom strictNormalFormsUnique :
  forall U V W : term, SNF U V -> SNF U W -> V = W.
```

### Normal terms

```
Axiom kNormal : Normal K.

Axiom kStrictlyNormal : SN K.
```

```
  Axiom ktNormal : forall T : term, Normal T -> Normal (K·T).

  Axiom ktStrictlyNormal : forall T : term, SN T -> SN (K·T).

  Axiom sNormal : Normal S.

  Axiom sStrictlyNormal : SN S.

  Axiom stNormal : forall T : term, Normal T -> Normal (S·T).

  Axiom stStrictlyNormal : forall T : term, SN T -> SN (S·T).

  Axiom suvNormal :
    forall U V : term,  Normal U -> Normal V -> Normal (S·U·V).

  Axiom suvStrictlyNormal : forall U V : term, SN U -> SN V -> SN (S·U·V).

  Axiom kuvNotStrictlyNormal : forall U V : term, ~ SN (K·U·V).

  Axiom suvwNotStrictlyNormal : forall U V W : term, ~ SN (S·U·V·W).
```

### Normal terms denote in all PCAs

```
  Axiom normalTermsDenote : forall `{_ : PCA.Pca} (T : term), Normal T -> §T!.
```

### Relation between strict reduction and closed term equivalence in PCAs

```
  Axiom eagerlyReduces_closedTermEq :
    forall `{_ : PCA.Pca} (U V : term), U >>* V -> §U ≃ §V.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Theorem strictlyNormalApplL (U V : term) : SN (U·V) -> SN U.
  Proof.
    intro H.
    remember (eagerReduct U) as rU eqn:HU. symmetry in HU.
    unfold SN in H. simpl in H.
    destruct rU as [U' | ]; rewrite HU in H.
    - discriminate H.
    - exact HU.
  Qed.

  Theorem strictlyNormalApplR (U V : term) : SN (U·V) -> SN V.
  Proof.
    intro H.
    remember (eagerReduct U) as rU eqn:HU.
    remember (eagerReduct V) as rV eqn:HV.
    symmetry in HU, HV. unfold SN in H. simpl in H.
    destruct rU as [U' | ]; rewrite HU in H.
```

```
  - discriminate H.
  - destruct rV as [V' | ].
    + rewrite HV in H. discriminate H.
    + exact HV.
Qed.

Theorem eagerReductionInduction (P : term -> term -> Prop) :
  (forall U U' V : term, P U U' -> P (U·V) (U'·V)) ->
  (forall U V V' : term, SN U -> P V V' -> P (U·V) (U·V')) ->
  (forall U V : term, SN U -> SN V -> P (K·U·V) U) ->
  (forall U V W : term, SN U -> SN V -> SN W -> P (S·U·V·W) (U·W·(V·W))) ->
  forall U V : term, U >> V -> P U V.
Proof.
  intros HL HR HK HS T.
  induction T as [ | | U IHU V IHV]; [discriminate | discriminate | ].
  remember (eagerReduct U) as rU eqn:HU. symmetry in HU.
  destruct rU as [U' | ].
  - intros W H.
    unfold EagerlyOneStepReduces in H. simpl in H. rewrite HU in H.
    injection H. intro H'. subst.
    apply HL. apply IHU. exact HU.
  - remember (eagerReduct V) as rV eqn:HV. symmetry in HV.
    destruct rV as [V' | ].
    + intros W H.
      unfold EagerlyOneStepReduces in H. simpl in H. rewrite HU, HV in H.
      injection H. intro H'. subst. apply HR.
      * exact HU.
      * apply IHV. exact HV.
    + intros W H.
      unfold EagerlyOneStepReduces in H. simpl in H. rewrite HU, HV in H.
      refine
        (match U as U' return (U' = U -> _) with
         | K       => _
         | S       => _
         | K·U'    => _
         | S·_     => _
         | K·_·_   => _
         | S·U'·V' => _
         | _·_·_·_ => _
         end _).
      * intro H'. subst. inversion H.
      * intro H'. subst. inversion H.
      * intro H'. subst. injection H. intro H'. subst.
        apply strictlyNormalApplR in HU. apply HK; assumption.
      * intro H'. subst. inversion H.
      * intro H'. subst. inversion H.
      * intro H'. subst. injection H. intro H'. subst.
        pose proof (strictlyNormalApplR _ _ HU) as HV'.
        apply strictlyNormalApplL, strictlyNormalApplR in HU.
        apply HS; assumption.
      * intro H'. subst. inversion H.
      * reflexivity.
Qed.

Theorem eagerOneStepReductionFunctional :
```

```
  RW.Functional EagerlyOneStepReduces.
Proof.
  unfold EagerlyOneStepReduces.
  intros U V W HUV HUW. rewrite HUV in HUW. injection HUW. tauto.
Qed.

Theorem eagerReductionConfluent : RW.Confluence EagerlyOneStepReduces.
Proof.
  apply RW.THMS.functionalRelation_confluence.
  apply eagerOneStepReductionFunctional.
Qed.

Theorem eagerlyOneStepReduces_oneStepReduces (U V : term) : U >> V -> U > V.
Proof.
  intro HUV.
  induction HUV as [U U' V IH | U V V' _ IH | U V _ _ | U V W _ _ _ ]
    using eagerReductionInduction.
  - apply oneStepReducesL. exact IH.
  - apply oneStepReducesR. exact IH.
  - apply oneStepReducesK.
  - apply oneStepReducesS.
Qed.

Theorem eagerlyReduces_reduces (U V : term) : U >>* V -> U >* V.
Proof.
  intro H. induction H as [U V | T | U V W _ IHL _ IHR].
  - apply rt_step. apply eagerlyOneStepReduces_oneStepReduces. exact H.
  - apply rt_refl.
  - apply rt_trans with (y := V); assumption.
Qed.

Theorem strictlyNormalCorrect (T : term) :
  SN T <-> RW.Normal EagerlyOneStepReduces T.
Proof.
  unfold RW.Normal, EagerlyOneStepReduces, SN.
  split.
  - intros H [T' H']. rewrite H' in H. discriminate H.
  - intros H. destruct (eagerReduct T) as [T' | T'].
    + exfalso. eauto.
    + reflexivity.
Qed.

Theorem strictNormalFormsCorrect (U V : term) :
  SNF U V <-> RW.NormalForm EagerlyOneStepReduces U V.
Proof.
  split.
  - intros [H HV]. split; [exact H | ]. clear U H. intros [W H].
    unfold SN in HV. unfold EagerlyOneStepReduces in H.
    rewrite HV in H. discriminate H.
  - intros [H HV]. split; [exact H | ]. clear U H.
    unfold SN.
    set (W' := eagerReduct V). remember W' as W eqn:HW.
    destruct W as [W | ]; subst.
    + exfalso. apply HV. unfold EagerlyOneStepReduces. eauto.
    + reflexivity.
```

```
Qed.

Theorem strictNormalFormsUnique (U V W : term) :
  SNF U V -> SNF U W -> V = W.
Proof.
  intros H H'.
  apply strictNormalFormsCorrect in H.
  apply strictNormalFormsCorrect in H'.
  eapply (RW.THMS.confluence_normalFormsUnique _ eagerReductionConfluent);
  eauto.
Qed.

Theorem normalApplL (U V : term) : Normal (U·V) -> Normal U.
Proof. intros H [T HT]. eauto using OneStepReduces. Qed.

Theorem normalApplR (U V : term) : Normal (U·V) -> Normal V.
Proof. intros H [T HT]. eauto using OneStepReduces. Qed.

Theorem normalReduction (U V : term) : Normal U -> U >* V -> U = V.
Proof.
  intros H H'.
  induction H' as [U V H' | U H' | U V W _ IHL _ IHR].
  - exfalso. eauto.
  - reflexivity.
  - specialize (IHL H). subst. exact (IHR H).
Qed.

Theorem strictlyNormalEagerReduction (U V : term) :
  SN U -> U >>* V -> U = V.
Proof.
  rewrite strictlyNormalCorrect. apply RW.THMS.normalReductionTrivial.
Qed.

Theorem kuvNotStrictlyNormal (U V : term) : ~ SN (K·U·V).
Proof.
  intro H.
  remember (eagerReduct U) as rU eqn:HU.
  remember (eagerReduct V) as rV eqn:HV.
  symmetry in HU, HV. unfold SN in H. simpl in H.
  destruct rU as [U' | ]; rewrite HU in H.
  - discriminate H.
  - destruct rV as [V' | ]; rewrite HV in H; discriminate H.
Qed.

Theorem suvwNotStrictlyNormal (U V W : term) : ~ SN (S·U·V·W).
Proof.
  intro H.
  remember (eagerReduct U) as rU eqn:HU.
  remember (eagerReduct V) as rV eqn:HV.
  remember (eagerReduct W) as rW eqn:HW.
  symmetry in HU, HV, HW. unfold SN in H. simpl in H.
  destruct rU as [U' | ]; rewrite HU in H.
  - discriminate H.
  - destruct rV as [V' | ]; rewrite HV in H.
    + discriminate H.
```

```
    + destruct rW as [W' | ]; rewrite HW in H; discriminate.
Qed.

Theorem normalIffStrictlyNormal (T : term) : Normal T <-> SN T.
Proof.
  split.
  - intro H. remember (eagerReduct T) as r eqn:Hr. symmetry in Hr.
    destruct r as [T' | ].
    + exfalso. apply H. exists T'.
      apply eagerlyOneStepReduces_oneStepReduces. exact Hr.
    + exact Hr.
  - intros HT [T' H]. induction H as [ | | U U' V _ IH | U V V' _ IH].
    + apply (kuvNotStrictlyNormal _ _ HT).
    + apply (suvwNotStrictlyNormal _ _ _ HT).
    + apply IH. eapply strictlyNormalApplL. eauto.
    + apply IH. eapply strictlyNormalApplR. eauto.
Qed.

Theorem strictNormalFormApplL (U V W : term) :
  SNF (U·V) W -> exists U' : term, SNF U U'.
Proof.
  intros [H HW]. remember (U·V) as T eqn:HT.
  revert U V HT.
  setoid_rewrite clos_rt_rt1n_iff in H.
  induction H as [T | T' T W HUVT HTW IH]; intros U V HT; subst.
  - exists U. split.
    + apply rt_refl.
    + eapply strictlyNormalApplL. eauto.
  - setoid_rewrite <- clos_rt_rt1n_iff in HTW. specialize (IH HW).
    remember (eagerReduct U) as U' eqn:HU.
    unfold EagerlyOneStepReduces in HUVT.
    destruct U' as [U' | ].
    + simpl in HUVT. rewrite <- HU in HUVT.
      injection HUVT. intro H. subst.
      specialize (IH U' V).
      destruct IH as [U'' [IH HU'']]; [reflexivity | ].
      exists U''. split; [ | exact HU'']. apply rt_trans with (y := U').
      * apply rt_step. symmetry. exact HU.
      * exact IH.
    + symmetry in HU. exists U. split.
      * apply rt_refl.
      * exact HU.
Qed.

Theorem strictNormalFormApplR (U V W : term) :
  SNF (U·V) W -> exists V' : term, SNF V V'.
Proof.
  intros [H HW]. remember (U·V) as T eqn:HT.
  revert U V HT.
  setoid_rewrite clos_rt_rt1n_iff in H.
  induction H as [T | T' T W HUVT HTW IH]; intros U V HT; subst.
  - exists V. split.
    + apply rt_refl.
    + eapply strictlyNormalApplR. eauto.
  - setoid_rewrite <- clos_rt_rt1n_iff in HTW. specialize (IH HW).
```

```
    remember (eagerReduct U) as U' eqn:HU.
    remember (eagerReduct V) as V' eqn:HV.
    unfold EagerlyOneStepReduces in HUVT.
    destruct U' as [U' | ]; [ | destruct V' as [V' | ]].
    + simpl in HUVT. rewrite <- HU in HUVT.
      injection HUVT. intro H. subst.
      eapply IH. eauto.
    + simpl in HUVT. rewrite <- HU, <- HV in HUVT.
      injection HUVT. intro H. subst.
      specialize (IH U V').
      destruct IH as [V'' [IH HV'']]; [reflexivity | ].
      exists V''. split; [ | exact HV'']. apply rt_trans with (y := V').
      * apply rt_step. symmetry. exact HV.
      * exact IH.
    + symmetry in HV. exists V. split.
      * apply rt_refl.
      * exact HV.
Qed.

Theorem strictNormalForm_normalForm (U V : term) : SNF U V -> NF U V.
Proof.
  firstorder using normalIffStrictlyNormal, eagerlyReduces_reduces.
Qed.

Theorem normalTermInduction (P : term -> Prop) :
  P K ->
  P S ->
  (forall T : term, Normal T -> P T -> P (K·T)) ->
  (forall T : term, Normal T -> P T -> P (S·T)) ->
  (forall U V : term, Normal U -> Normal V -> P U -> P V -> P (S·U·V)) ->
  forall T : term, Normal T -> P T.
Proof.
  assert (forall T U V W, ~ Normal (T·U·V·W)) as lemma. {
    clear. intro T.
    induction T as [ | | U IHU V _].
    - intros U V W H. apply H. exists (U·W). constructor. constructor.
    - intros U V W H. apply H. eexists. constructor.
    - intros W U' V' H. apply (IHU V W U'). eapply normalApplL. eauto.
  }
  intros HK HS HKU HSU HSUV T.
  refine
    ((fix IH T :=
     match T as T' return (T = T' -> Normal T ->  P T) with
     | K       => _
     | S       => _
     | K·T     => _
     | S·T     => _
     | K·U·V   => _
     | S·U·V   => _
     | _·_·_·_ => _
     end _) T).
  - intros H _. subst. apply HK.
  - intros H _. subst. apply HS.
  - intros H' H. subst.
    pose proof (normalApplR _ _ H) as HV.
```

```
    apply (HKU _ HV). apply (IH _ HV).
  - intros H' H. subst.
    pose proof (normalApplR _ _ H) as HV.
    apply (HSU _ HV). apply (IH _ HV).
  - intros H' H. subst. exfalso. apply H. exists U. constructor.
  - intros H' H. subst.
    pose proof (normalApplL _ _ H) as HU. apply normalApplR in HU.
    pose proof (normalApplR _ _ H) as HV.
    apply (HSUV _ _ HU HV).
    + apply (IH _ HU).
    + apply (IH _ HV).
  - intros H' H. subst. exfalso. eapply lemma. eauto.
  - reflexivity.
Qed.

Theorem strictlyNormalTermInduction (P : term -> Prop) :
  P K ->
  P S ->
  (forall T : term, SN T -> P T -> P (K·T)) ->
  (forall T : term, SN T -> P T -> P (S·T)) ->
  (forall U V : term, SN U -> SN V -> P U -> P V -> P (S·U·V)) ->
  forall T : term, SN T -> P T.
Proof.
  setoid_rewrite <- normalIffStrictlyNormal. apply normalTermInduction.
Qed.

Theorem eagerReductionL (U U' V : term) : U >>* U' -> U·V >>* U'·V.
Proof.
  intros H. induction H as [U U' H | U | U T U' _ IHL _ IHR].
  - apply rt_step.
    unfold EagerlyOneStepReduces. simpl. rewrite H.
    reflexivity.
  - apply rt_refl.
  - eapply rt_trans; eauto.
Qed.

Theorem eagerReductionR (U V V' : term) : SN U -> V >>* V' -> U·V >>* U·V'.
Proof.
  intros HU H. induction H as [V V' H | V | V T V' _ IHL _ IHR].
  - apply rt_step.
    unfold EagerlyOneStepReduces. simpl. rewrite HU, H.
    reflexivity.
  - apply rt_refl.
  - eapply rt_trans; eauto.
Qed.

Theorem eagerReductionPar (U U' V V' : term) :
  SN U' -> U >>* U' -> V >>* V' -> U·V >>* U'·V'.
Proof.
  intros HU' HU HV. apply rt_trans with (y := U'·V).
  - apply eagerReductionL. exact HU.
  - apply eagerReductionR; assumption.
Qed.

Theorem reductionL (U U' V : term) : U >* U' -> U·V >* U'·V.
```

```
Proof.
  intro H. induction H as [U U' | U | U W U' HL IHL HR IHR].
  - apply rt_step. constructor. assumption.
  - apply rt_refl.
  - apply rt_trans with (y := W·V); assumption.
Qed.

Theorem reductionR (U V V' : term) :  V >* V' -> U·V >* U·V'.
Proof.
  intro H. induction H as [V V' | V | V W V' HL IHL HR IHR].
  - apply rt_step. constructor. assumption.
  - apply rt_refl.
  - apply rt_trans with (y := U·W); assumption.
Qed.

Theorem reductionPar (U V U' V' : term) :
  U >* U' -> V >* V' -> U·V >* U'·V'.
Proof.
  intros HU HV.
  apply rt_trans with (y := U'·V);
  [apply reductionL | apply reductionR]; assumption.
Qed.

Theorem kStrictlyNormal : SN K.
Proof. reflexivity. Qed.

Theorem kNormal : Normal K.
Proof. apply normalIffStrictlyNormal, kStrictlyNormal. Qed.

Theorem ktStrictlyNormal (T : term) : SN T -> SN (K·T).
Proof. unfold StrictlyNormal. simpl. intro H. rewrite H. reflexivity. Qed.

Theorem ktNormal (T : term) : Normal T -> Normal (K·T).
Proof. setoid_rewrite normalIffStrictlyNormal. apply ktStrictlyNormal. Qed.

Theorem sStrictlyNormal : SN S.
Proof. reflexivity. Qed.

Theorem sNormal : Normal S.
Proof. apply normalIffStrictlyNormal, sStrictlyNormal. Qed.

Theorem stStrictlyNormal (T : term) : SN T -> SN (S·T).
Proof. unfold StrictlyNormal. simpl. intro H. rewrite H. reflexivity. Qed.

Theorem stNormal (T : term) : Normal T -> Normal (S·T).
Proof. setoid_rewrite normalIffStrictlyNormal. apply stStrictlyNormal. Qed.

Theorem suvStrictlyNormal (U V : term) : SN U -> SN V -> SN (S·U·V).
Proof.
  unfold SN. simpl. intros H H'. rewrite H, H'. reflexivity.
Qed.

Theorem suvNormal (U V : term) : Normal U -> Normal V -> Normal (S·U·V).
Proof. setoid_rewrite normalIffStrictlyNormal. apply suvStrictlyNormal. Qed.
```

```
Local Reserved Infix ">=" (at level 70).
Inductive OneStepReflParReduces : term -> term -> Prop :=
| oneStepReflParReducesK    U V       : K·U·V >= U
| oneStepReflParReducesS    U V W     : S·U·V·W >= U·W·(V·W)
| oneStepReflParReducesRefl U         : U >= U
| oneStepReflParReducesPar  U U' V V' : U >= U' -> V >= V' -> U·V >= U'·V'
where "U >= V" := (OneStepReflParReduces U V).

Definition ReflParReduces := RTC OneStepReflParReduces.
Local Infix ">=*" := ReflParReduces (at level 70).

Lemma inclusion1 (U V : term) : U > V -> U >= V.
Proof. intro H. induction H; auto using OneStepReflParReduces. Qed.

Lemma inclusion2 (U V : term) : U >= V -> U >* V.
Proof.
  intro H. induction H as [V T | U V W | V | U U' V V' HL IHL HR IHR].
  - apply rt_step. constructor.
  - apply rt_step. constructor.
  - apply rt_refl.
  - apply rt_trans with (y := U'·V);
    [apply reductionL | apply reductionR]; assumption.
Qed.

Lemma redEqReflParRed (U V : term) : U >* V <-> U >=* V.
Proof.
  split.
  - apply RW.THMS.rtcIncreasing. apply inclusion1.
  - intros H.
    induction H as [U V H | U | U W V _ IHL _ IHR].
    + apply inclusion2. assumption.
    + apply rt_refl.
    + apply rt_trans with (y := W); assumption.
Qed.

Lemma dpOneStepReflParReduces : DP OneStepReflParReduces.
Proof.
  (* redundant cases *)
  assert (forall T U V, K·T >= U -> exists W, T >= W /\ U·V >= W) as case1.
  {
    intros T U V H. inversion_clear H.
    - eauto using OneStepReflParReduces.
    - inversion_clear H0. eauto using OneStepReflParReduces.
  }
  assert
    (forall U V W U' V',
     S·U·V >= U' -> W >= V' -> exists W', U·W·(V·W) >= W' /\ U'·V' >= W')
  as case2. {
    intros U V W U' V' HL HR. remember (S·U·V) as SUv eqn:HSUV.
    destruct HL as [U'' V'' | U'' V'' W'' | T | U'' U''' V'' V''' HL' HR'].
    - inversion HSUV.
    - discriminate HSUV.
    - subst. eauto 6 using OneStepReflParReduces.
    - inj_subst HSUV. inversion_clear HL'.
      + eauto 6 using OneStepReflParReduces.
```

```
      + inversion_clear H. eauto 6 using OneStepReflParReduces.
  }
  (* the proof *)
  intros T U V H. generalize dependent V.
  induction H as [U V | U V W | T | U U' V V' HL IHL HR IHR].
  - remember (K·U·V) as T eqn:HT.
    intros W H.
    destruct H as [U' V' | U' V' W' | T | U'' U' V'' V' H' H'' ].
    + inj_subst HT. eauto using OneStepReflParReduces.
    + inversion HT.
    + subst. eauto using OneStepReflParReduces.
    + inj_subst HT. apply (case1 _ _ _ H').
  - remember (S·U·V·W) as T eqn:HT.
    intros T' H.
    destruct H as [U' V' | U' V' W' | T | U'' U' V'' V' H' H''].
    + inversion HT.
    + inj_subst HT. eauto using OneStepReflParReduces.
    + subst. eauto using OneStepReflParReduces.
    + inj_subst HT. apply (case2 _ _ _ _  _ H' H'').
  - eauto using OneStepReflParReduces.
  - remember (U·V) as T eqn:HT.
    intros T' H.
    destruct H as [U'' V'' | U'' V'' W'' | T | U''' U'' V''' V'' HL' HR'].
    + inj_subst HT. pose proof (case1 _ _ V' HL). firstorder.
    + inj_subst HT. pose proof (case2 _ _ _ _ _ HL HR). firstorder.
    + subst. eauto using OneStepReflParReduces.
    + inj_subst HT.
      destruct
        (IHL _ HL') as [U''' [H H']],
        (IHR _ HR') as [V''' [H'' H''']].
      eauto using OneStepReflParReduces.
Qed.

Theorem reductionConfluent : RW.Confluence OneStepReduces.
Proof.
  cut (DP ReflParReduces).
  + intros H U V V' H' H''.
    apply redEqReflParRed in H'.
    apply redEqReflParRed in H''.
    destruct (H _ _ _ H' H'') as [W [H''' H4]].
    apply redEqReflParRed in H'''.
    apply redEqReflParRed in H4.
    eauto.
  + apply RW.THMS.dp_confluence. apply dpOneStepReflParReduces.
Qed.

Theorem normalFormsUnique (U V V' : term) : NF U V -> NF U V' -> V = V'.
Proof.
  intros H H'.
  eapply (RW.THMS.confluence_normalFormsUnique _ reductionConfluent); eauto.
Qed.

Instance convertibleEquivalence : Equivalence Convertible.
Proof.
  split.
```

```
    - intro T. exists T. split; apply rt_refl.
    - firstorder.
    - intros U V w [U' [HUV HVU] [V' [HVW HWV]]].
      destruct (reductionConfluent _ _ _ HVU HVW) as [W' [HU'W' HV'W']].
      exists W'. split; eapply rt_trans; eauto.
Qed.

Instance normalFormsRespectConvertibility :
  Proper (Convertible ==> eq ==> iff) NF.
Proof.
  assert (forall U V W, U ≈ V -> NF U W -> NF V W) as H. {
    intros U V W [T [HUT HVT]] [HUW HW].
    destruct (reductionConfluent _ _ _ HUT HUW) as [W' [HTW' HWW']].
    apply (normalReduction _ _ HW) in HWW'. subst. split.
    - eapply rt_trans; eauto.
    - exact HW.
  }
  intros U V HUV W' W HW. subst.
  split; [ | symmetry in HUV]; apply H; exact HUV.
Qed.

Instance applRespectsConvertibility :
  Proper (Convertible ==> Convertible ==> Convertible) appl.
Proof.
  intros U U' [U'' [HU HU']] V V' [V'' [HV HV']]. exists (U''·V'').
  split; apply reductionPar; assumption.
Qed.

Theorem normalTermsDenote `{pca : PCA.Pca} (T : term) : Normal T -> §T!.
Proof.
  revert T. apply normalTermInduction.
  - exists k. reflexivity.
  - exists s. reflexivity.
  - intros U _ [a Ha]. simpl. rewrite Ha. apply PCA.THMS.kaDenotes.
  - intros U _ [a Ha]. simpl. rewrite Ha. apply PCA.THMS.saDenotes.
  - intros U V _ _ [a Ha] [b Hb]. simpl. rewrite Ha, Hb. apply PCA.sSpec1.
Qed.

Theorem eagerlyReduces_closedTermEq `{pca : PCA.Pca} (U V : term) :
  U >>* V -> §U ≃ §V.
Proof.
  intro H. apply clos_rt_rt1n in H.
  induction H as [T | U V W HL _ IH]; [reflexivity | ].
  rewrite <- IH. clear dependent W.
  induction HL as [U U' V IH | U V V' H IH | U V HU HV | U V W _ _ _ ]
    using eagerReductionInduction;
  simpl.
  - rewrite IH. reflexivity.
  - rewrite IH. reflexivity.
  - apply normalIffStrictlyNormal in HU.
    apply normalIffStrictlyNormal in HV.
    destruct
      (normalTermsDenote _ HU) as [a Ha],
      (normalTermsDenote _ HV) as [b Hb].
    rewrite Ha, Hb. apply PCA.kSpec.
```

```
      - apply PCA.THMS.sSpec2'.
    Qed.

  End THMS.

End KS_RW.
```

# L. termpca.v—Partial combinatory algebra based on KS-terms

```
Require Import Coq.Classes.Morphisms.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Relations.Relation_Operators.

Require Import fin.
Require Import ksrewriting.
Require Import pas.
Require Import pca.

Import KS_RW.
Import KS_RW.NOTATIONS.
Import KS_RW.THMS.
Import PAS.NOTATIONS.

Local Open Scope PAS.
Local Open Scope KS_RW.

Module TERM_PCA.
```

## Definitions

```
  Instance termSetoid : Setoid term := {equiv := Convertible}.

  Definition Appl : term -> term -> term -> Prop :=
    fun U V W => U·V == W.

  Fixpoint closedPasTermToTerm `{_ : PAS.Pas term} (t : PAS.term 0) : term :=
  match t with
  | &a        => a
  | PAS.var x => False_rect _ (FIN.THMS.t0Empty x)
  | u*v       => closedPasTermToTerm u · closedPasTermToTerm v
  end.
  Local Notation "$ t" := (closedPasTermToTerm t) (at level 6).
```

## Notations

```
  Module NOTATIONS.

    Delimit Scope TERM_PCA with TERM_PCA.

    Notation "$ t" := ($t) (at level 6) : TERM_PCA.

  End NOTATIONS.
```

## Theorems

```
Module Type THMS_SIG.

  Declare Instance pas : PAS.Pas termSetoid Appl.

  Axiom totality : PAS.Total pas.

  Declare Instance pca : PCA.Pca pas K S.

  Axiom nonTriviality : PCA.NonTrivial pca.

  Declare Instance applRespectsEquiv :
    Proper (equiv ==> equiv ==> equiv) appl.

  Declare Instance closedPasTermToTermRespectsEquiv :
    Proper (PAS.ClosedTermEq ==> equiv) closedPasTermToTerm.

  Axiom closedPasTermToTermEq : forall t : PAS.term 0, t ≃ &$t.

  Axiom closedPasTermToTermInjective : forall u v : PAS.term 0,
    $u == $v -> u ≃ v.

End THMS_SIG.
```

## Proofs

```
Module THMS : THMS_SIG.

  Instance applRespectsEquiv : Proper (equiv ==> equiv ==> equiv) appl :=
    applRespectsConvertibility.

  Instance pas : PAS.Pas termSetoid Appl.
  Proof.
    split; unfold Appl.
    - intros u u' v v' w w' Hu Hv Hw Huv.
      rewrite <- Hu, <- Hv, <- Hw. exact Huv.
    - intros u v w w' H H'. rewrite <- H, H'. reflexivity.
  Qed.

  Theorem totality : PAS.Total pas.
  Proof. intros u v. exists (u·v). simpl. unfold Appl. reflexivity. Qed.

  Theorem closedPasTermToTermEq (t : PAS.term 0) : t ≃ &$t.
  Proof.
    induction t as [t | x | u IHu v IHv].
    - reflexivity.
    - contradiction.
    - simpl. rewrite IHu, IHv at 1. rewrite PAS.THMS.correctness1'.
      unfold Appl. reflexivity.
  Qed.

  Instance closedPasTermToTermRespectsEquiv :
    Proper (PAS.ClosedTermEq ==> equiv) closedPasTermToTerm.
  Proof.
```

```
      intros u v H.
      setoid_rewrite closedPasTermToTermEq in H.
      apply PAS.THMS.constInjective in H. rewrite H. reflexivity.
    Qed.

    Theorem closedPasTermToTermInjective (u v : PAS.term 0) : §u == §v -> u ≃ v.
    Proof.
      setoid_rewrite closedPasTermToTermEq. simpl.
      intro H. rewrite H. reflexivity.
    Qed.

    Instance pca : PCA.Pca pas K S.
    Proof.
      split.
      - intros a b. apply closedPasTermToTermInjective. simpl. exists a. split.
        + apply rt_step. apply oneStepReducesK.
        + apply rt_refl.
      - intros a b. exists (S·a·b). apply closedPasTermToTermInjective. simpl.
        exists (S·a·b). split; apply rt_refl.
      - intros a b c. apply closedPasTermToTermInjective. simpl.
        exists (a·c·(b·c)). split.
        + apply rt_step. apply oneStepReducesS.
        + apply rt_refl.
    Qed.

    Theorem nonTriviality : PCA.NonTrivial pca.
    Proof.
      exists K, S. intros [t [HK HS]].
      apply (normalReduction _ _ kNormal) in HK.
      apply (normalReduction _ _ sNormal) in HS.
      subst. discriminate HS.
    Qed.

  End THMS.

End TERM_PCA.
```

# M. normaltermpca.v—Partial combinatory algebra based on normal KS-terms

```
Require Import Coq.Classes.Morphisms.
Require Import Coq.Classes.SetoidClass.
Require Import Coq.Relations.Operators_Properties.
Require Import Coq.Relations.Relation_Operators.

Require Import fin.
Require Import ksrewriting.
Require Import pas.
Require Import pca.
Require Import rewriting.

Import KS_RW.
Import KS_RW.NOTATIONS.
```

```
Import KS_RW.THMS.
Import PAS.NOTATIONS.
Import PCA.NOTATIONS.

Local Open Scope PAS.
Local Open Scope KS_RW.

Module NORMAL_TERM_PCA.
```

## Definitions

```
  Definition normalTerm := {t : term | StrictlyNormal t}.

  Definition K : normalTerm := exist _ _ kStrictlyNormal.
  Definition S : normalTerm := exist _ _ sStrictlyNormal.

  Definition normalTermToTerm (T : normalTerm) : term := proj1_sig T.
  Local Coercion normalTermToTerm : normalTerm >-> term.

  Definition NormalTermEq (U V : normalTerm) := (U : term) = (V : term).

  Instance normalTermEqEquivalence : Equivalence NormalTermEq.
  Proof. unfold NormalTermEq. split; congruence. Qed.

  Instance normalTermSetoid : Setoid normalTerm := {equiv := NormalTermEq}.

  Definition Appl (U V W : normalTerm) : Prop := U·V >>* W.
```

## Theorems

```
  Module Type THMS_SIG.

    Declare Instance pas : PAS.Pas normalTermSetoid Appl.

    Declare Instance pca : PCA.Pca pas K S.

    Axiom nonTriviality : PCA.NonTrivial pca.

    Axiom normalTermDenotation : forall (T : normalTerm), §T ≃ &T.

    Axiom termDenotation :
      forall (U : term) (V : normalTerm), §U ≃ &V <-> U >>* V.

  End THMS_SIG.
```

## Proofs

```
  Module THMS : THMS_SIG.

    Instance pas : PAS.Pas normalTermSetoid Appl.
    Proof.
      split.
      - unfold Appl. congruence.
      - unfold Appl.
```

```
    intros [U HU] [V HV] [W HW] [W' HW'] H H'. simpl in *.
    unfold NormalTermEq. simpl.
    eapply strictNormalFormsUnique; unfold SNF; eauto.
Qed.

Fixpoint termToClosedPasTerm (T : term) : PAS.term 0 :=
match T with
| KS_RW.K   => &K
| KS_RW.S   => &S
| U·V       => termToClosedPasTerm U * termToClosedPasTerm V
end.
Local Notation "$ t" := (termToClosedPasTerm t) (at level 6).

Lemma termDenotation' (U : term) (V : normalTerm) :
  $(U) ≃ &V <-> U >>* V.
Proof.
  revert V. induction U as [ | | U IHU V IHV]; intro T.
  - simpl.
    assert (&K ≃ &T <-> K == T) as H. {
      split; intro H.
      - apply PAS.THMS.constInjective. exact H.
      - rewrite H. reflexivity.
    }
    rewrite H. clear H. simpl. unfold NormalTermEq. split; intro H.
    + rewrite <- H. simpl. apply rt_refl.
    + destruct T as [T HT]. simpl in *.
      apply strictlyNormalEagerReduction.
      * apply kStrictlyNormal.
      * exact H.
  - simpl.
    assert (&S ≃ &T <-> S == T) as H. {
      split; intro H.
      - apply PAS.THMS.constInjective. exact H.
      - rewrite H. reflexivity.
    }
    rewrite H. clear H. simpl. unfold NormalTermEq. split; intro H.
    + rewrite <- H. simpl. apply rt_refl.
    + destruct T as [T HT]. simpl in *.
      apply strictlyNormalEagerReduction.
      * apply sStrictlyNormal.
      * exact H.
  - split; intro H.
    + simpl in H. apply PAS.THMS.correctness1 in H.
      destruct H as [U' [V' [HU [HV H]]]].
      unfold Appl in H.
      specialize (IHU U'). destruct IHU as [IHU _]. specialize (IHU HU).
      specialize (IHV V'). destruct IHV as [IHV _]. specialize (IHV HV).
      clear HU HV. apply rt_trans with (y := U'·V').
      * destruct U' as [U' HU'], V' as [V' HV']. simpl in *.
        apply eagerReductionPar; assumption.
      * exact H.
    + simpl.
      assert (SNF (U·V) T) as H'. {
        split.
        - exact H.
```

```
        - destruct T as [T HT]. exact HT.
      }
      destruct
        (strictNormalFormApplL _ _ _ H') as [U' [HU HU']],
        (strictNormalFormApplR _ _ _ H') as [V' [HV HV']].
      clear H'.
      set (U'' := exist _ _ HU' : normalTerm).
      set (V'' := exist _ _ HV' : normalTerm).
      specialize (IHU U''). destruct IHU as [_ IHU]. rewrite (IHU HU).
      specialize (IHV V''). destruct IHV as [_ IHV]. rewrite (IHV HV).
      rewrite PAS.THMS.correctness1'. unfold Appl. simpl.
      clear IHU IHV U'' V''.
      pose proof (eagerReductionPar _ _ _ _ HU' HU HV) as H'.
      destruct
        (RW.THMS.functionalDoubleReduction
          EagerlyOneStepReduces eagerOneStepReductionFunctional _ _ _ H H')
      as [H'' | H''];
      [ | exact H''].
      destruct T as [T HT]. simpl in *.
      apply (strictlyNormalEagerReduction _ _ HT) in H''. subst.
      apply rt_refl.
Qed.

Lemma normalTermDenotation' (T : normalTerm) : $(T) ≃ &T.
Proof. setoid_rewrite termDenotation'. apply rt_refl. Qed.

Lemma eagerlyReduces_closedPasTermEq (U V : term) :
  U >>* V -> $(U) ≃ $(V).
Proof.
  intro H. rewrite PAS.THMS.correctness2.
  setoid_rewrite termDenotation'.
  intros [T HT]. simpl. split; intro H'.
  - destruct (eagerReductionConfluent _ _ _ H H') as [T' [H'' HT']].
    apply (strictlyNormalEagerReduction _ _ HT) in HT'.
    subst. exact H''.
  - eapply rt_trans; eauto.
Qed.

Instance pca : PCA.Pca pas K S.
Proof.
  split.
  - intros U V.
    setoid_rewrite <- normalTermDenotation'.
    cutrewrite ($(K)*$(U)*$(V) = $(K·U·V)); [ | reflexivity].
    apply eagerlyReduces_closedPasTermEq.
    destruct U as [U HU], V as [V HV]. simpl.
    apply rt_step. unfold EagerlyOneStepReduces. simpl. rewrite HU, HV.
    reflexivity.
  - intros U V.
    setoid_rewrite <- normalTermDenotation'.
    cutrewrite ($(S)*$(U)*$(V) = $(S·U·V)); [ | reflexivity].
    unfold PAS.Denotes. setoid_rewrite termDenotation'.
    destruct U as [U HU], V as [V HV]. simpl.
    set (SUV := exist _ _ (suvStrictlyNormal _ _ HU HV) : normalTerm).
    exists SUV. apply rt_refl.
```

```
      - intros U V W.
        setoid_rewrite <- normalTermDenotation'.
        cutrewrite ($(S)*$(U)*$(V)*$(W) = $(S·U·V·W)); [ | reflexivity].
        cutrewrite ($(U)*$(W)*($(V)*$(W)) = $(U·W·(V·W))); [ | reflexivity].
        apply eagerlyReduces_closedPasTermEq.
        destruct U as [U HU], V as [V HV], W as [W HW]. simpl.
        apply rt_step.
        unfold EagerlyOneStepReduces. simpl. rewrite HU, HV, HW.
        reflexivity.
    Qed.

    Theorem nonTriviality : PCA.NonTrivial pca.
    Proof. exists K, S. intros H. discriminate H. Qed.

    Lemma equalMaps (T : term) : §T = $(T).
    Proof.
      induction T as [ | | U IHU V IHV].
      - reflexivity.
      - reflexivity.
      - simpl. rewrite IHU, IHV. reflexivity.
    Qed.

    Theorem normalTermDenotation (T : normalTerm) : §T ≃ &T.
    Proof. rewrite equalMaps. apply normalTermDenotation'. Qed.

    Theorem termDenotation (U : term) (V : normalTerm) : §U ≃ &V <-> U >>* V.
    Proof. rewrite equalMaps. apply termDenotation'. Qed.

  End THMS.

End NORMAL_TERM_PCA.
```

## Coercions

```
Coercion NORMAL_TERM_PCA.normalTermToTerm :
  NORMAL_TERM_PCA.normalTerm >-> KS_RW.term.
```

## References


Beeson, Michael (1985). *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer.

Friedberg, Richard M. (1958). "Three theorems on recursive enumeration. I. Decomposition. II. Maximal set. III. Enumeration without duplication". In: *Journal of Symbolic Logic* 23.3, pp. 309–316.

Jacobs, Bart (1999). *Categorical Logic and Type Theory*. Elsevier.

Kleene, Stephen Cole (1945). "On the interpretation of intuitionistic number theory". In: *Journal of Symbolic Logic* 10.4, pp. 109–124.

Martin-Löf, Per (1971). "A theory of types". Preprint. Department of Mathematics, Stockholm University.

Nelson, David (1947). "Recursive functions and intuitionistic number theory". In: *Transactions of the American Mathematical Society* 61.2, pp. 307–368.

Pollack, Robert (1995). "Polishing up the Tait–Martin-Löf proof of the Church–Rosser theorem". In: *Proceedings of De Wintermöte '95*. Department of Computing Science, Chalmers University of Technology.

Soare, Robert Irving (1999). *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Springer.

Troelstra, Anne Sjerp and Helmut Schwichtenberg (2000). *Basic Proof Theory*. 2nd ed. Cambridge University Press.

Van Oosten, Jaap (2008). *Realizability: An Introduction to its Categorical Side*. Elsevier.

TRITA -MAT-E 2015:71
ISRN -KTH/MAT/E--15/71--SE