Standardization and Confluence in Pure λ-Calculus Formalized for the Matita Theorem Prover

Ferruccio Guidi
Department of Computer Science and Engineering
University of Bologna
Mura Anteo Zamboni 7, 40127, Bologna, ITALY
e-mail: fguidi@cs.unibo.it

We present a formalization of pure λ-calculus for the Matita interactive theorem prover, including the proofs of two relevant results in reduction theory: the confluence theorem and the standardization theorem. The proof of the latter is based on a new approach recently introduced by Xi and refined by Kashima that, avoiding the notion of development and having a neat inductive structure, is particularly suited for formalization in theorem provers.

1. INTRODUCTION

Standardization and confluence are two fundamental properties of β-reduction sequences in pure λ-calculus [Bar85]. In order to state these properties, we will denote λ-terms with the capital letters A, B, C, D, M, N, and will agree that by a *computation* we mean a reduction sequence. In particular, a *sequential* computation is such that a single redex is contracted in every step.

The standardization theorem asserts that if M computes to N, then there is computation from M to N which is *standard* in that redexes contracted in each step are not "left" residuals of redexes contracted in the previous steps.

The confluence theorem, also known as the Church-Rosser property, asserts that if M computes to N_1 and to N_2, then there is an N which both N_1 and N_2 compute to. This is to say that N_1 and N_2 have a common reduct.

We remark that by "λ-calculus" we presently mean the $\lambda K\beta$-calculus.

Here we discuss how we formalized the proofs of these propositions for the Matita interactive theorem prover [ARST11], with the final aim of providing a set of relevant results on pure λ-calculus to the HELM digital library [APS$^+$03].

Matita uses the Calculus of Inductive Constructions [CP90] as metalanguage, in a variant similar to the one underlying the Coq proof management system [Coq12]. We tried to present things in a such a way that a limited knowledge of Matita's specification language, both in its syntactical and semantical aspects, is required.

As to confluence, we follow the proof of Tait and Matin-Löf based on parallel reduction [Tak95], which has already been formalized in several logical frameworks (see for instance [Pfe92]). This proof was chosen because it does not require to extend the pure λ-calculus with marks for tracing redexes.

As to standardization, we follow essentially the proof of Xi [Xi99] in the particularly clear presentation made by Kashima [Kas00]. This novel approach entirely avoids delicate notions such as *residual* or *development* and is entirely based on structural induction on λ-terms and derivations, resulting particularly appealing for formalization in interactive provers. Previous formal proofs of standardization, like [MP99], are sensibly more entangled than the one we shall describe here.

In order to define the notion of standard derivation, Kashima associates to each redex a natural number, termed its *degree* in [RP04], corresponding to its position in a "left-to-right" visit of the term. This system of reference is a bit inconvenient in the perspective of a formalization: firstly it requires an ancillary function to compute the number of redexes occurring in a term, secondly it forces to add at least two clauses to the definition of the relation "M reduces to N contracting one redex that is referred as n" with respect to the case in which n is not specified.

To solve these inconveniences, we adopt a system of reference by *pointers* that are paths in the tree representation of terms, for the specification of which, strings of boolean values suffice. Following the ideas of Kashima, pointers can be given an order reflecting the standard reduction order as defined by means of residuals, and that is equivalent to the "left-to-right" order induced by degrees.

Our exposition is organized as follows. Section 2 contains the part of our formalization that is not directly related to λ-calculus. As a formal system, pure λ-calculus has two components, which we discuss in Section 3 and Section 4. The *structural* component comprises the description of terms, with α-conversion, and with substitution. The *behavioral* component includes reduction and computation. Conversion is not considered in this work. Our conclusions are in Section 5.

The formalization we are presenting in this article takes advantage of our long-time experience in the machine-checked specification of λ-calculus [Gui06] acquired by developing the formal system $\lambda\delta$ [Gui09] since 2004.

2. BACKGROUND AND PRINCIPLES OF THE FORMALIZATION

Our formalization is for Matita 0.99.1 and depends on some notions and facts contained in the library provided with the prover. In particular we use logical connectives, existential quantifiers (the universal quantifier is in the metalanguage), Leibniz equality, general properties of relations especially concerning reflexive and transitive closures, arithmetics of natural numbers, and monomorphic lists.

In this section we discuss the additions we felt necessary in order to facilitate the formalization process. Some of them are now included in the library.

Of the many tactics provided by Matita's proof engine, we use the following for our proofs: intro, lapply, apply, elim, rewrite, cases, destruct, normalize, auto.

In particular, we manage to avoid generalize, and inversion. Our aversion for the latter tactic, comes from Coq 7, which generates oversize proof terms by inlining an inversion lemma in the proof at each invocation of the tactic. For instance, Lemma `pr0_confluence`[1] of [Gui06] contains 49 invocations of inversion producing a proof that, as a tree, has 46103 nodes and takes more than six minutes to validate on an AMD Athlon MP 1800+ at 1.5 GHz.

A newer and still unpublished version of the proof, produced without invoking inversion, is 29% smaller than the former and consists of just 32593 nodes.

Moreover the inversion of Coq does not invert some recursive relations properly since all inductive premises generated by the inversion procedure are removed from

[1]<http://lambdadelta.info/static/matita/basic_1/pr0/pr0/pr0_confluence.con.body.html>.

the proof context. Nevertheless, this is not the expected behavior if one inverts the type judgment of a λ-calculus that includes the type conversion rule.

Matita's inversion does not inline an inversion lemma and keeps the inductive premises in place, yet we prefer to encapsulate the inversion proof steps in *inversion lemmas* that follow the style of the well-known "inversion lemma" for Pure Type Systems [Bar93] and that we prove by a combination of cases and destruct.

Furthermore, we hide types as much as possible following [Gui10]. For this reason we avoid the tactics that they take types as parameters.

We make a substantial use of auto to automate backward reasoning. Matita automation tactics takes a few optional parameters to prune the search space. The most relevant ones are *depth* and *width*: the first one is the maximum number of applicative nodes in a branch of the proof, while the second one is the maximum number of *mutually dependent* open goals along each branch.

Having experienced dramatic delays in the execution of auto due to the large span of the proofs search space even at low depths, some precautions must be adopted.

On one hand, most propositions that look problematic in view of a backward automatic synthesis of the proof should not be indexed for automation. These include some transitive-like properties of relations and the propositions that are designed to be applied in a forward reasoning manner as the inversion lemmas.

On the other hand we limit the search space depth by preferring flat n-ary logical constructions over nested ones of fixed arity. This is to say that the proposition:

$$\exists x_1,\ldots,x_n.P_1(x_1,\ldots,x_n)\wedge\ldots\wedge P_m(x_1,\ldots,x_n)$$

is formalized with a single multiple existential quantifier rather than with:

$$\exists x_1.(\ldots(\exists x_n.(\ldots(P_1(x_1,\ldots,x_n)\wedge\ldots)\ldots)\wedge P_m(x_1,\ldots,x_n))\ldots)$$

that is, by means of the standard existential quantifier and binary conjunction.

In the first form, the proposition is atomized at depth 1, i.e. by the backward application of one constructor, whereas in the second form, the proposition is atomized at depth $m+n-1$, i.e. by applying $m+n-1$ constructors.

To this aim, we implemented an application that generates Matita's scripts with the definitions of multiple existential quantifies for a given set of pairs (m,n).

We stress that the case $(6,6)$ is realistic in the formalization of λ-calculus (see Lemma `pr0_gen_appl`[2] and Lemma `pr2_gen_appl`[3] of [Gui06]).

The existential quantifier $(2,1)$ is in Matita's library. In this formalization we also use the instances: $(2,2)$, $(2,3)$, $(3,2)$, $(3,3)$, $(4,3)$, and a ternary disjunction.

2.1 Logic and Arithmetics

We use the following property of the existential quantifier $(2,1)$ in the proof of the confluence theorem. Our multiple existential quantifiers are denoted by ∃∃.

```
lemma ex2_commute: ∀A0. ∀P0,P1:A0→Prop.
                   (∃∃x0. P0 x0 & P1 x0) → ∃∃x0. P1 x0 & P0 x0.
```

[2] <http://lambdadelta.info/static/matita/basic_1/pr0/fwd/pr0_gen_appl.con.html>.
[3] <http://lambdadelta.info/static/matita/basic_1/pr2/fwd/pr2_gen_appl.con.html>.

We decided to avoid the logical negation (¬) and thus we activated the notation for the logical contradiction (⊥) that is not active in Matita's library.

The following propositions use this notation and our machine-generated ternary disjunction connective (∨∨) in the context of natural arithmetics:

```
lemma lt_refl_false : ∀n. n < n → ⊥.

lemma lt_zero_false: ∀n. n < 0 → ⊥.

lemma plus_lt_false: ∀m,n. m + n < m →⊥.

lemma lt_or_eq_or_gt: ∀m,n. ∨∨ m < n | n = m | n < m.
```

We also use the *trichotomy operator* `tri` returning a_1, a_2, or $a3$ depending on n_1 being less, equal, or greater than n_2. This function could be defined on top of the library, but we believe that a direct definition is easier to deal with:

```
let rec tri (A:Type[0]) n1 n2 a1 a2 a3 on n1 : A ≝
  match n1 with
  [ O    ⇒ match n2 with [ O ⇒a2 | S n2 ⇒ a1 ]
  | S n1 ⇒ match n2 with [ O ⇒a3 | S n2 ⇒ tri A n1 n2 a1 a2 a3 ]
  ].

lemma tri_lt: ∀A,a1,a2,a3,n2,n1. n1 < n2 → tri A n1 n2 a1 a2 a3 = a1.

lemma tri_eq: ∀A,a1,a2,a3,n. tri A n n a1 a2 a3 = a2.

lemma tri_gt: ∀A,a1,a2,a3,n1,n2. n2 < n1 → tri A n1 n2 a1 a2 a3 = a3.
```

Finally we need a general induction principle on a type A based on weight f:

```
lemma f_ind_aux: ∀A. ∀f:A→nat. ∀P:predicate A.
                   (∀n. (∀a. f a < n → P a) → ∀a. f a = n → P a) →
                   ∀n,a. f a = n → P a.

lemma f_ind: ∀A. ∀f:A→nat. ∀P:predicate A.
               (∀n. (∀a. f a < n → P a) → ∀a. f a = n → P a) → ∀a. P a.
```

2.2 Relations and Lists

The reflexive and transitive closure of a relation is in the library:

```
inductive star (A:Type[0]) (R:relation A) (a:A): A → Prop ≝
  |sstep: ∀b,c.star A R a b → R b c → star A R a c
  | srefl : star A R a a.
```

and comes with some standard properties about it.

We added the eliminator `star_ind_l`, that is not provided by Matita:

```
lemma star_ind_l: ∀A,R,a2. ∀P:predicate A.
                    P a2 →
                    (∀a1,a. R a1 a → star ... R a a2 → P a → P a1) →
                    ∀a1. star ... R a1 a2 → P a1.
```

We also added a general notion of a confluent relation (see Subsection 4.4).

We show, through the well-known "strip" lemma, that the reflexive and transitive closure of a confluent relation is confluent. This result will be used in Subsection 4.5.

```
definition singlevalued: ∀A,B. predicate (relation2  A B) ≝ λA,B,R.
                         ∀a,b1.  R a b1 → ∀b2. R a b2 → b1 = b2.

definition confluent1: ∀A. relation  A → predicate A ≝ λA,R,a0.
                       ∀a1. R a0 a1 → ∀a2. R a0 a2 →
                       ∃∃a. R a1 a & R a2 a.

definition confluent: ∀A. predicate (relation  A) ≝ λA,R.
                      ∀a0. confluent1 ... R a0.

lemma star_strip: ∀A,R. confluent A R →
                  ∀a0,a1. star ... R a0 a1 → ∀a2. R a0 a2 →
                  ∃∃a. R a1 a & star ... R a2 a.

lemma star_confluent: ∀A,R. confluent A R →confluent  A (star ... R).
```

A *labeled sequential reduction* (see Subsection 4.3) is a binary relation on terms taking an extra argument and needing a dedicated reflexive and transitive closure:

```
inductive lstar (A:Type[0]) (B:Type[0]) (R: A→relation  B): list  A →  relation B ≝
|  lstar_nil  : ∀b. lstar  A B R ([]) b b
|  lstar_cons : ∀a,b1,b. R a b1 b →
                ∀l,b2. lstar  A B R l b b2 →  lstar  A B R (a::l) b1 b2
.

definition  ltransitive : ∀A,B:Type[0]. predicate ( list  A →  relation  B) ≝ λA,B,R.
                          ∀l1,b1,b. R l1 b1 b → ∀l2,b2. R l2 b b2 → R (l1@l2) b1 b2.
```

The notion `lstar` comes with the standard properties derived from those of `star`. In addition, we proved the following:

```
lemma lstar_inv_nil: ∀A,B,R,l,b1,b2. lstar  A B R l b1 b2 →  []  = l  →  b1 = b2.

lemma lstar_inv_cons: ∀A,B,R,l,b1,b2. lstar  A B R l b1 b2 →
                      ∀a0,l0. a0::l0  = l  →
                      ∃∃b. R a0 b1 b & lstar A B R l0 b b2.

lemma lstar_inv_step: ∀A,B,R,a,b1,b2. lstar  A B R ([a]) b1 b2  →  R a b1 b2.

lemma lstar_inv_pos: ∀A,B,R,l,b1,b2. lstar  A B R l b1 b2 →  0 < |l|  →
                     ∃∃a,l1,b.  a:: l1  = l & R a b1 b & lstar A B R l1 b b2.
```

Lemma `lstar_inv_pos` seems to be one of the deep grounds of the standardization theorem as we understand from Lemma `st_lsred_swap` of Subsection 4.7.

We activated an alternative notation for the empty list (◇), because the notation provided by the library ([ ]) clashes with our notation for substitution.

Finally, we added a variant of `All` acting on a binary predicate (see Subsection 3.6) and a function `map_cons` whose application is denoted by $a ::: l$:

```
let rec Allr (A:Type[0]) (R:relation A) (l: list A) on l : Prop ≝
match l with
[ nil       ⇒ True
| cons a1 l ⇒ match l with [ nil ⇒ True | cons a2 _ ⇒ R a1 a2 ∧ Allr A R l ]
].

definition map_cons: ∀A. A → list (list A) → list (list A) ≝ λA,a.
                      map ... (cons ... a).
```

3. THE STRUCTURAL COMPONENT OF PURE λ-CALCULUS

In this section we describe the structure of pure λ-terms and two related topics: the support for α-conversion and for substitution. The last subsection is devoted to the location of subterms through paths on the tree representation of λ-terms.

All the material in this section is relatively standard, and we only include it to introduce the notation and for the sake of completeness.

3.1 Pure λ-Terms

Pure λ-terms are inductively generated from variable references (`VRef`), function formations (`Abst`), and function applications (`Appl`):

```
inductive term: Type[0] ≝
| VRef: nat  → term
| Abst: term → term
| Appl: term → term → term
.
```

Variable references occur by depth [dB94] for a convenient management of α-conversion and reduction. In particular, with this encoding two α-convertible terms are syntactically equal. Our depths, also known as "de Bruijn indexes", are natural numbers starting at 0 rather than at 1. So all depths are legal and they will be typically ranged over by the metavariables i and j.

The formation of the function with body M is denoted by $\mathrm{\lambda}.M$, in which the character $\mathrm{\lambda}$ is typeset in upright face (contrast with the character λ).

The application of the function M to the argument N is denoted here by $@N.M$ according to the "reversed" notation suggested in [KN96], that, among other benefits, avoids precedence problems and keeps the matching abstraction-application pairs close together improving the visual understanding of redexes.

Then we define some notions of compatibility with term constructors and we prove that these notions are preserved by reflexive and transitive closures.

```
definition compatible_abst: predicate (relation term) ≝ λR.
                             ∀A1,A2. R A1 A2 →R (λ.A1) (λ.A2).

definition compatible_sn: predicate (relation term) ≝ λR.
                           ∀A,B1,B2. R B1 B2 →R (@B1.A) (@B2.A).
```

```
definition compatible_dx: predicate (relation term) ≝ λR.
                              ∀B,A1,A2. R A1 A2 →R (@B.A1) (@B.A2).

definition compatible_appl: predicate (relation term) ≝ λR.
                              ∀B1,B2. R B1 B2 →∀A1,A2. R A1 A2 →
                              R (@B1.A1) (@B2.A2).

lemma star_compatible_abst: ∀R. compatible_abst R →compatible_abst (star ... R).

lemma star_compatible_sn: ∀R. compatible_sn R →compatible_sn (star ... R).

lemma star_compatible_dx: ∀R. compatible_dx R →compatible_dx (star ...R).

lemma star_compatible_appl: ∀R. reflexive ? R →
                              compatible_appl R → compatible_appl (star ... R).
```

3.2 Relocation

The management of de Bruijn indexes requires the well-known function `lift`, here denoted by $\uparrow[d,h]$, that is related to the function τ_h of [dB94]. Here and in the following d and e denote a level whereas h and k denote a relocation offset. The function `lift` extends to terms the function ${}_d{\uparrow}^h$: `nat` → `nat` acting on indexes:

$$ {}_d{\uparrow}^h\, i \overset{\text{def}}{=} \begin{cases} i & \text{if } i < d \\ i+h & \text{if } i \geq d \end{cases} $$

By applying this function some indexes are increased by the offset h (*lifting*). The inverse operation, which decreases some indexes, is named *delifting.* Our theory of `lift` is taken from [Gui06] and can be traced back to [Hue94].

```
let rec lift h d M on M ≝match M with
[ VRef i   ⇒ #(tri ... i d i (i + h) (i + h))
| Abst A   ⇒ λ. (lift h (d+1) A)
| Appl B A ⇒ @(lift h d B). (lift h d A)
].

lemma lift_vref_lt: ∀d,h,i. i < d → ↑[d, h] #i = #i.

lemma lift_vref_ge: ∀d,h,i. d ≤ i → ↑[d, h] #i = #(i+h).

lemma lift_id: ∀M,d. ↑[d, 0] M = M.
```

We provide the following inversion lemmas for `lift`:

```
lemma lift_inv_vref_lt: ∀j,d. j < d → ∀h,M. ↑[d, h] M = #j →M = #j.

lemma lift_inv_vref_ge: ∀j,d. d ≤ j → ∀h,M. ↑[d, h] M = #j →
                        d + h ≤ j ∧ M = #(j−h).

lemma lift_inv_vref_be: ∀j,d,h. d ≤ j → j < d + h → ∀M. ↑[d, h] M = #j →⊥.
lemma lift_inv_vref_ge_plus: ∀j,d,h. d + h ≤ j →
                             ∀M. ↑[d, h] M = #j → M = #(j−h).

lemma lift_inv_abst: ∀C,d,h,M. ↑[d, h] M = λ.C →
                     ∃∃A. ↑[d+1, h] A = C & M = λ.A.

lemma lift_inv_appl: ∀D,C,d,h,M. ↑[d, h] M = @D.C →
                     ∃∃B,A. ↑[d, h] B = D & ↑[d, h] A = C & M = @B.A.
```

Then we compute the compound relocation ${}_{d_2}{\uparrow}^{h_2}\,{}_{d_1}{\uparrow}^{h_1}$ in the three cases: $d_2 \le d_1$ ("less or equal"), $d_1 \le d_2 \le d_1 + h_1$ ("between"), $d_1 + h_1 \le d_2$ ("greater or equal").
The proofs are by induction on the structure of `M`.

```
theorem lift_lift_le : ∀h1,h2,M,d1,d2. d2 ≤ d1 →
                       ↑[d2, h2] ↑[d1, h1] M = ↑[d1 + h2, h1] ↑[d2, h2] M.

theorem lift_lift_be : ∀h1,h2,M,d1,d2. d1 ≤ d2 → d2 ≤ d1 + h1 →
                       ↑[d2, h2] ↑[d1, h1] M = ↑[d1, h1 + h2] M.

theorem lift_lift_ge : ∀h1,h2,M,d1,d2. d1 + h1 ≤ d2 →
                       ↑[d2, h2] ↑[d1, h1] M = ↑[d1, h1] ↑[d2 − h1, h2] M.
```

The inverse propositions of the above are also provided.
The proofs are by induction on the structure of `M1`.

```
theorem lift_inj: ∀h,M1,M2,d. ↑[d, h] M2 = ↑[d, h] M1 → M2 = M1.
#h #M1 elim M1 −M1

theorem lift_inv_lift_le : ∀h1,h2,M1,M2,d1,d2. d2 ≤ d1 →
                           ↑[d2, h2] M2 = ↑[d1 + h2, h1] M1 →
                           ∃∃M. ↑[d1, h1] M = M2 & ↑[d2, h2] M = M1.

theorem lift_inv_lift_be : ∀h1,h2,M1,M2,d1,d2. d1 ≤ d2 → d2 ≤ d1 + h1 →
                           ↑[d2, h2] M2 = ↑[d1, h1 + h2] M1 → ↑[d1, h1] M1 = M2.

theorem lift_inv_lift_ge : ∀h1,h2,M1,M2,d1,d2. d1 + h1 ≤ d2 →
                           ↑[d2, h2] M2 = ↑[d1, h1] M1 →
                           ∃∃M. ↑[d1, h1] M = M2 & ↑[d2 − h1, h2] M = M1.
```

In the end we define the notion of stability under the application of `lift` and we prove that stability is preserved by reflexive and transitive closures.

```
definition liftable : predicate (relation term) ≝ λR.
                      ∀h,M1,M2. R M1 M2 → ∀d. R (↑[d, h] M1) (↑[d, h] M2).

definition deliftable_sn : predicate (relation term) ≝ λR.
                           ∀h,N1,N2. R N1 N2 → ∀d,M1. ↑[d, h] M1 = N1 →
                           ∃∃M2. R M1 M2 & ↑[d, h] M2 = N2.

lemma star_liftable: ∀R. liftable R → liftable (star … R).
lemma star_deliftable_sn: ∀R. deliftable_sn R → deliftable_sn (star ... R).

lemma lstar_liftable: ∀T,R. (∀t. liftable (R t)) →
                      ∀l. liftable (lstar T ... R l).

lemma lstar_deliftable_sn: ∀T,R. (∀t. deliftable_sn (R t)) →
                           ∀l. deliftable_sn (lstar T ... R l).
```

3.3 Delifting Substitution

When we operate with de Bruijn indexes, substituting N in M means replacing the occurrences of the first free variable in M with N. The commonly accepted version of substitution we are presenting here, also delifts the occurrences of the other free variables in M. This practice makes the theory of substitution not so elegant, but removes the delifting operation from the definition of reduction.

We remark that both [dB94] and [Gui09] do not follow this practice.

The substitution of N for the occurrences of the first free variable at level d in M (`dsubst`) is denoted here by $[d \swarrow\!\!\!\!\nearrow N]\, M$, where the slanted arrow reminds that some delifting is performed. Other widespread notations include $M\,[N/d]$ and $M\,[d \leftarrow N]$. If the parameter d is missing, the value 0 is implied for it.

As for `lift`, we take our theory of `dsubst` from [Gui06] and [Hue94]. Given that `dsubst` is not injective, its theory is much shorter than the one of `lift`.

```
let rec dsubst D d M on M ≝ match M with
[ VRef i    ⇒ tri ... i d (#i) (↑[i] D) (#(i−1))
| Abst A    ⇒ λ. (dsubst D (d+1) A)
| Appl B A ⇒ @ (dsubst D d B). (dsubst D d A)
].

lemma dsubst_vref_lt: ∀i,d,D. i < d → [d ↙↗D] #i = #i.

lemma dsubst_vref_eq: ∀i,D. [i ↙↗D] #i = ↑[i]D.

lemma dsubst_vref_gt: ∀i,d,D. d < i → [d ↙↗D] #i = #(i−1).
```

The theory includes the composition of `dsubst` at level d_2 after `lift` at level d_1 in the three cases considered in Subsection 3.2.

The proofs are by induction on the structure of the first premise.

```
theorem dsubst_lift_le: ∀h,D,M,d1,d2. d2 ≤ d1 →
                        [d2 ↙↗↑[d1 − d2, h] D] ↑[d1 + 1, h] M = ↑[d1, h] [d2 ↙↗D] M.

theorem dsubst_lift_be: ∀h,D,M,d1,d2. d1 ≤ d2 → d2 ≤ d1 + h →
                        [d2 ↙↗D] ↑[d1, h + 1] M = ↑[d1, h] M.

theorem dsubst_lift_ge: ∀h,D,M,d1,d2. d1 + h ≤ d2 →
                        [d2 ↙↗D] ↑[d1, h] M = ↑[d1, h] [d2 − h ↙↗D] M.
```

The theory also includes the composition of `dsubst` at level d_2 after `dsubst` at level d_1 in the cases $d2 < d1$ ("less than") and $d1 \leq d2$ ("greater or equal").

Lemma `dsubst_dsubst_ge` is the well-known "substitution lemma", which is proved here by induction on the structure of its first premise.

```
theorem dsubst_dsubst_ge: ∀D1,D2,M,d1,d2. d1 ≤d2 →
                          [d2 ↙D2] [d1 ↙D1] M =
                          [d1 ↙[d2 − d1 ↙D2] D1] [d2 + 1 ↙D2] M.

theorem dsubst_dsubst_lt: ∀D1,D2,M,d1,d2. d2 < d1 →
                          [d2 ↙[d1 − d2 −1 ↙D1] D2] [d1 ↙D1] M =
                          [d1 − 1 ↙D1] [d2 ↙D2] M.
```

The theory ends with some preservation results about the notion of stability under substitution in the spirit of Subsection 3.2.

```
definition dsubstable_dx: predicate (relation term) ≝ λR.
                          ∀D,M1,M2. R M1 M2 →∀d. R ([d ↙D] M1) ([d ↙D] M2).

definition dsubstable: predicate (relation term) ≝ λR.
                       ∀D1,D2. R D1 D2 →∀M1,M2. R M1 M2 →
                       ∀d. R ([d ↙D1] M1) ([d ↙D2] M2).

lemma star_dsubstable_dx: ∀R. dsubstable_dx R →dsubstable_dx (star ... R).

lemma lstar_dsubstable_dx: ∀T,R. (∀t. dsubstable_dx (R t)) →
                           ∀l. dsubstable_dx (lstar T ... R l).

lemma star_dsubstable: ∀R. reflexive ? R →
                       dsubstable R → dsubstable (star ... R).
```

3.4 Size

In this formalization, the *size* of a term M, denoted by $|M|$, is the number of inner nodes in the tree representation of M. In the literature (see for instance [Xi99]) the "size" of M may refer to the total numbers of nodes in M as well.

We need this notion in the present work to set up a well-founded induction in the proof of the so-called "diamond property" (see Subsection 4.4).

The proof of the lemma is by induction on the structure of `M`.

```
let rec size M on M ≝match M with
[ VRef i   ⇒ 0
| Abst A   ⇒ size A + 1
| Appl B A ⇒ (size B) + (size A) + 1
].

lemma size_lift: ∀h,M,d. |↑[d, h] M| = |M|.
```

3.5 Pointers to Subterms

In this formalization we wish to qualify a subterm N of a term M with the path that connects the root of M to the root of N in the tree representation of M. We will refer to such a path as to the *pointer* to N. Clearly, the case of N being a redex is of particular importance in the following sections.

A pointer, ranged over by the metavariables p and q, is a list of steps and c will be a metavariable ranging over steps. We recognize three steps named `rc` ("rectum" or "straight"), `sn` ("sinister" or "left") and `dx` ("dexter" or "right"). We note that `rc` and `sn` can be identified as long as head reductions [Bar85] are not considered.

The subterm of M pointed by a step and by a list of steps is defined as follows:

(1) `rc` points to A in $\lambda.A$;

(2) `sn` points to B and `dx` points to A in $@B.A$;

(3) $\diamondsuit$ points to M in M itself;

(4) if c points to N_1 in M and p points to N_2 in N_1, then $c :: p$ points to N_2 in M.

A term M with a pointer in in bijection with an approximation [RP04] of M with a single instance of the term Ω.

Steps (`ptr_step`) and pointers (`ptr`) are formally defined in the following lines with the notions of "compatibility with λ-constructions" concerning relations.

```
inductive ptr_step: Type[0] ≝
| rc: ptr_step
| sn: ptr_step
| dx: ptr_step
.

definition ptr: Type[0] ≝ list ptr_step.

definition compatible_rc: predicate (ptr → relation term) ≝ λR.
                          ∀p,A1,A2. R p A1 A2 →R (rc::p) (λ.A1) (λ.A2).

definition compatible_sn: predicate (ptr → relation term) ≝ λR.
                          ∀p,B1,B2,A. R p B1 B2 →R (sn::p) (@B1.A) (@B2.A).

definition compatible_dx: predicate (ptr → relation term) ≝ λR.
                          ∀p,B,A1,A2. R p A1 A2 →R (dx::p) (@B.A1) (@B.A2).
```

The theory ends with the definition of the predicate `in_whd` selecting the pointers made just of `dx` steps, which locate the "weak head redex" of a term. An elimination principle for reasoning about these pointers is provided.

Its proof is by induction on the structure of `p`.

```
definition is_dx: predicate ptr_step ≝ λc. dx = c.

definition in_whd: predicate ptr ≝ All ... is_dx.

lemma in_whd_ind: ∀R:predicate ptr. R (◇) →
                  (∀p. in_whd p → R p → R (dx::p)) →
                  ∀p. in_whd p → R p.
```

3.6 Lists of Pointers to Subterms

In this formalization we use lists of pointers to trace the redexes contracted in finite sequential computations. Thus a computation is as long as its list of pointers.

Lists of pointers, ranged over by the metavariables r and s, are defined in the following lines and come with compatibility notions and results (see Subsection 3.5).

definition ptrl: Type[0] $\overset{\text{def}}{=}$ list ptr.

definition ho_compatible_rc: predicate (ptrl → relation term) $\overset{\text{def}}{=}$ λR.
∀s,A1,A2. R s A1 A2 →R (rc ::: s) (λ.A1) (λ.A2).

definition ho_compatible_sn: predicate (ptrl → relation term) $\overset{\text{def}}{=}$ λR.
∀s,B1,B2,A. R s B1 B2 →R (sn ::: s) (@B1.A) (@B2.A).

definition ho_compatible_dx: predicate (ptrl → relation term) $\overset{\text{def}}{=}$ λR.
∀s,B,A1,A2. R s A1 A2 →R (dx ::: s) (@B.A1) (@B.A2).

lemma lstar_compatible_rc: ∀R. compatible_rc R →ho_compatible_rc (lstar ... R).

lemma lstar_compatible_sn: ∀R. compatible_sn R →ho_compatible_sn (lstar ... R).

lemma lstar_compatible_dx: ∀R. compatible_dx R →ho_compatible_dx (lstar ... R).

The predicate `is_whd` selects the lists of `in_whd` pointers, which we name *whd* lists. These lists are meant to trace weak head reduction sequences [Pey87].

The two proofs are by induction on the structure of s and r respectively.

definition is_whd: predicate ptrl $\overset{\text{def}}{=}$ All ... in_whd.

lemma is_whd_dx: ∀s. is_whd s → is_whd (dx ::: s).

lemma is_whd_append: ∀r. is_whd r →∀s. is_whd s → is_whd (r@s).

4. THE BEHAVIORAL COMPONENT OF PURE λ-CALCULUS

In this section we formalize β-reduction and the related notion of β-reduction sequence (or β-computation) both in the "labeled sequential" and "parallel" forms.

The section starts by introducing an order on pointers modeling the "leftmost outermost" (or "left-to-right") reduction strategy for the pointed redexes.

4.1 Standard Order on Pointers to Subterms

The order on pointers we will define in this section is meant to model the order in which the pointed redexes are contracted in the "leftmost outermost" (or "left-to-right") reduction strategy underlying the standardization theorem.

This order, which is our alternative to the order of [Kas00], is based on a precedence relation $\prec$ (`pprec`) defined in the following lines.

The meaning of $p \prec q$ is: first contract the redex pointed by p in a given term M and then the redex pointed by q in the contractum of M.

```
inductive pprec: relation ptr ≝
| pprec_nil  : ∀c,q.    pprec (◇) (c :: q)
| pprec_rc   : ∀p,q.    pprec (dx :: p) (rc :: q)
| pprec_sn   : ∀p,q.    pprec (rc :: p) (sn :: q)
| pprec_comp: ∀c,p,q. pprec p q → pprec (c :: p) (c :: q)
| pprec_skip:           pprec (dx :: ◇) ◇
.

lemma pprec_inv_sn: ∀p,q. p ≺ q → ∀p0. sn :: p0 = p →
                     ∃∃q0. p0 ≺ q0 & sn::q0 = q.
```

We justify our definition as follows. With `pprec_nil` we state that if M is itself a redex, it must be contracted before other redexes. With `pprec_rc` we state that redexes in functions must be contracted before redexes in function bodies. With `pprec_sn` we state that redexes in function bodies must be contracted before redexes in function arguments. With `pprec_comp` we state that the order must be preserved when contracting inner redexes. With `pprec_skip` we state that contracting a redex in a function may produce a redex to be contracted afterwards.

The *standard order* $\leq$ (`ple`) is defined as the reflexive and transitive closure of the precedence relation, and comes with standard properties.

```
definition ple: relation ptr ≝ star … pprec.

lemma ple_step_rc: ∀p,q. p ≺ q → p ≤ q.

lemma ple_step_sn: ∀p1,p,p2. p1 ≺ p → p ≤ p2 → p1 ≤ p2.

lemma ple_rc: ∀p,q. dx::p ≤ rc :: q.

lemma ple_sn: ∀p,q. rc::p ≤ sn :: q.

lemma ple_skip: dx::◇ ≤ ◇.

lemma ple_nil: ∀p. ◇ ≤ p.

lemma ple_comp: ∀p1,p2. p1 ≤p2 → ∀c. (c :: p1) ≤ (c :: p2).

lemma ple_skip_ple: ∀p. p ≤ ◇ → dx :: p ≤ ◇.

theorem ple_trans: transitive … ple.

lemma ple_cons: ∀p,q. dx::p ≤ sn :: q.

lemma ple_dichotomy: ∀p1,p2:ptr. p1 ≤ p2 ∨ p2 ≤ p1.

lemma ple_inv_sn: ∀p,q. p ≤ q → ∀p0. sn :: p0 = p →
                   ∃∃q0. p0 ≤ q0 & sn::q0 = q.
```

This statement characterizes the `in_whd` pointers in terms of the order:

The following are equivalent for p: `in_whd` p; $p \leq \diamondsuit$; $\forall q.\ p \leq q$.

The proofs are by induction on the structure of the first premise.

lemma pprec_fwd_in_whd: $\forall$p,q. p $\prec$q $\rightarrow$ in_whd q $\rightarrow$ in_whd p.

lemma in_whd_ple_nil: $\forall$p. in_whd p $\rightarrow$ p $\leq$ $\diamondsuit$.

theorem in_whd_ple: $\forall$p. in_whd p $\rightarrow\forall$q. p $\leq$ q.

lemma ple_nil_inv_in_whd: $\forall$p. p $\leq$ $\diamondsuit$ $\rightarrow$ in_whd p.

lemma ple_inv_in_whd: $\forall$p. ($\forall$q. p $\leq$ q) $\rightarrow$ in_whd p.

The fact that the standard order is cyclic, in that $\diamondsuit \prec$ `dx` :: $\diamondsuit \prec \diamondsuit$, is not surprising and is due to the existence of cyclic computations in λ-calculus.

One can derive from this chain of inequalities that the transitive closure of the precedence relation is reflexive (just remember Clause `pprec_comp`).

4.2 Standard Lists of Pointers to Subterms

The predicate `is_standard` selects the lists of pointers occurring in non-decreasing standard order, which we name *standard* lists. According to the justification of the standard precedence relation we gave in Subsection 4.1, it is plausible that these lists trace standard reduction sequences. As one expects, Lemma `is_whd_is_standard` asserts that a whd list is standard as is a weak head reduction sequence.

The proofs are by induction on the structure of `r` or `s`.

definition is_standard: predicate ptrl $\overset{\text{def}}{=}$ Allr ... ple.

lemma is_standard_compatible: $\forall$c,s. is_standard s $\rightarrow$ is_standard (c ::: s).

lemma is_standard_cons: $\forall$p,s. is_standard s $\rightarrow$ is_standard ((dx::p)::sn ::: s).

lemma is_standard_append: $\forall$r. is_standard r $\rightarrow$ $\forall$s. is_standard s $\rightarrow$
is_standard ((dx ::: r)@sn:::s).

theorem is_whd_is_standard: $\forall$s. is_whd s $\rightarrow$ is_standard s.

lemma is_standard_in_whd: $\forall$p. in_whd p $\rightarrow\forall$s. is_standard s $\rightarrow$ is_standard (p::s).

theorem is_whd_is_standard_trans: $\forall$r. is_whd r $\rightarrow$ $\forall$s. is_standard s $\rightarrow$
is_standard (r@s).

We justify informally our notion of standard list in terms of residuals as follows. Take a term C with a two redexes M and N pointed by p and q respectively. If M is is a residual at the "left" of N, then reducing M after N is not standard and we argue that $q \leq p$ is forbidden by the order in the most relevant cases.

If $C \equiv M \equiv @B.\lambda.A$, then N is a subterm of B or of A, which is to say that q has the form `sn` :: q_0 or `dx` :: `rc` :: q_0. Now $q \leq p$ and $p \equiv \diamondsuit$ imply `in_whd` q.

Now take M and N in the arguments of $C \equiv @D.@B.A$, so M is a subterm of B and N is a subterm of D (remember that C is $(A\ B\ D)$ in standard notation). So q has the form `sn` :: q_0 and p has the form `dx` :: `sn` :: p_0 in contrast with Lemma `ple_inv_sn` by which $q \leq p$ yields that p must have the form `sn` :: p_1.

4.3 Labeled Sequential Reduction

The advantage of our path-based redex pointers over the degree-based ones used in [Kas00], clearly appears when (finally!) we come to define a step of sequential reduction labeled with the pointer to the redex we are contracting.

This relation (`lsred`), which we denote with $M \mapsto[p]\ N$, comes with standard properties such as stability under relocation and substitution. In addition, some inversion lemmas prove that the relation is single-valued for given p and M.

```
inductive lsred: ptr → relation term ≝
| lsred_beta    : ∀B,A. lsred (◇) (@B.λ.A) ([↙B]A)
| lsred_abst    : ∀p,A1,A2. lsred p A1 A2 → lsred (rc :: p) (λ.A1) (λ.A2)
| lsred_appl_sn : ∀p,B1,B2,A. lsred p B1 B2 → lsred (sn :: p) (@B1.A) (@B2.A)
| lsred_appl_dx : ∀p,B,A1,A2. lsred p A1 A2 → lsred (dx :: p) (@B.A1) (@B.A2)
.

lemma lsred_inv_vref: ∀p,M,N. M ↦[p] N →∀i. #i = M →⊥.

lemma lsred_inv_nil: ∀p,M,N. M ↦[p] N →◇= p →
                     ∃∃B,A. @B. λ.A = M & [↙B] A = N.

lemma lsred_inv_rc: ∀p,M,N. M ↦[p] N →∀q. rc::q = p →
                    ∃∃A1,A2. A1 ↦[q] A2 & λ.A1 = M & λ.A2 = N.

lemma lsred_inv_sn: ∀p,M,N. M ↦[p] N →∀q. sn::q = p →
                    ∃∃B1,B2,A. B1 ↦[q] B2 & @B1.A = M & @B2.A = N.

lemma lsred_inv_dx: ∀p,M,N. M ↦[p] N →∀q. dx::q = p →
                    ∃∃B,A1,A2. A1 ↦[q] A2 & @B.A1 = M & @B.A2 = N.

lemma lsred_lift: ∀p. liftable (lsred p).

lemma lsred_inv_lift: ∀p. deliftable_sn (lsred p).

lemma lsred_dsubst: ∀p. dsubstable_dx (lsred p).

theorem lsred_mono: ∀p. singlevalued ...(lsred p).
```

4.4 Parallel Reduction

The purpose of parallel reduction in this formalization is to support the confluence of sequential computation by providing its well-known "diamond" property.

One step of parallel reduction (`pred`), which we denote with $M \Mapsto N$, contracts an arbitrary set of redexes that do not need to be traced for our present aims.

This relation comes with standard properties such as reflexivity and stability under relocation and substitution. Some inversion lemmas are provided as well.

```
inductive pred: relation term ≝
| pred_vref: ∀i. pred (#i) (#i)
| pred_abst: ∀A1,A2. pred A1 A2 →pred (λ.A1) (λ.A2)
| pred_appl: ∀B1,B2,A1,A2. pred B1 B2 →pred A1 A2 →pred (@B1.A1) (@B2.A2)
| pred_beta: ∀B1,B2,A1,A2. pred B1 B2 →pred A1 A2 →pred (@B1.λ.A1) ([↙B2]A2)
.

lemma pred_refl: reflexive … pred.

lemma pred_inv_vref: ∀M,N. M ⤇N →∀i. #i = M →#i = N.

lemma pred_inv_abst: ∀M,N. M ⤇N →∀A. λ.A = M →
                     ∃∃C. A ⤇ C & λ.C = N.

lemma pred_inv_appl: ∀M,N. M ⤇N →∀B,A. @B.A = M →
                     (∃∃D,C. B ⤇ D & A ⤇ C & @D.C = N) ∨
                     ∃∃A0,D,C0. B ⤇D & A0 ⤇ C0 & λ.A0 = A & [↙D]C0 = N.

lemma pred_lift: liftable pred.

lemma pred_inv_lift: deliftable_sn pred.

lemma pred_dsubst: dsubstable pred.

lemma lsred_pred: ∀p,M,N. M ↦[p] N →M ⤇N.
```

The confluence of parallel reduction (also known as its “diamond” property), stating that $M_0 \Mapsto M_1$ and $M_0 \Mapsto M_2$ imply $M_1 \Mapsto M$ and $M_2 \Mapsto M$ for some M, is proved by induction on the size of M_0. The induction generates six cases, three of which are proved as separate lemmas because one of them is invoked twice.

In this way we take advantage of the symmetry existing between M_1 and M_2.

```
lemma pred_conf1_vref: ∀i. confluent1 … pred (#i).
#i #M1 #H1 #M2 #H2
<(pred_inv_vref … H1) −H1 [3: // |2: skip ]
<(pred_inv_vref … H2) −H2 [3: // |2: skip ]
/2 width=3/
qed−.

lemma pred_conf1_abst: ∀A. confluent1 …pred A → confluent1 … pred (λ.A).
#A #IH #M1 #H1 #M2 #H2
elim (pred_inv_abst … H1 ??) −H1 [3: // |2: skip ] #A1 #HA1 #H destruct
elim (pred_inv_abst … H2 ??) −H2 [3: // |2: skip ] #A2 #HA2 #H destruct
elim (IH … HA1 …HA2) −A /3 width=3/
qed−.
```

```
lemma pred_conf1_appl_beta: ∀B,B1,B2,C,C2,M1.
                             (∀M0. |M0| < |B|+|λ.C|+1 →confluent1 ? pred M0) →
                             B ⤇ B1 → B ⤇ B2 → λ.C ⤇ M1 → C ⤇ C2 →
                             ∃∃M. @B1.M1 ⤇M & [↙B2]C2 ⤇M.
#B #B1 #B2 #C #C2 #M1 #IH #HB1 #HB2 #H1 #HC2
elim (pred_inv_abst … H1 ??) −H1 [3: // |2: skip ] #C1 #HC1 #H destruct
elim (IH B … HB1 … HB2) −HB1 −HB2 //
elim (IH C … HC1 … HC2) normalize // −B −C /3 width=5/
qed−.

theorem pred_conf: confluent … pred.
#M @(f_ind … length … M) −M #n #IH ∗ normalize
[ /2 width=3 by pred_conf1_vref/
| /3 width=4 by pred_conf1_abst/
| #B #A #H #M1 #H1 #M2 #H2 destruct
  elim (pred_inv_appl … H1 ???) −H1 [5: // |2,3: skip ] ∗
  elim (pred_inv_appl … H2 ???) −H2 [5,10: // |2,3,7,8: skip ] ∗
  [ #B2 #A2 #HB2 #HA2 #H2 #B1 #A1 #HB1 #HA1 #H1 destruct
    elim (IH A … HA1 … HA2) −HA1 −HA2 //
    elim (IH B … HB1 … HB2) // −A −B /3 width=5/
  | #C #B2 #C2 #HB2 #HC2 #H2 #HM2 #B1 #N #HB1 #H #HM1 destruct
    @(pred_conf1_appl_beta … IH) //
  | #B2 #N #B2 #H #HM2 #C #B1 #C1 #HB1 #HC1 #H1 #HM1 destruct
    @ex2_commute @(pred_conf1_appl_beta … IH) //
  | #C #B2 #C2 #HB2 #HC2 #H2 #HM2 #C0 #B1 #C1 #HB1 #HC1 #H1
    #HM1 destruct
    elim (IH B … HB1 … HB2) −HB1 −HB2 //
    elim (IH C … HC1 … HC2) normalize // −B −C /3 width=5/
  ]
]
qed−.
```

4.5 Parallel Computation

Parallel computation (`preds`) from M to N is denoted by $M \Rrightarrow^* N$ and is defined as the reflexive and transitive closure of parallel reduction. This notion comes with standard properties including confluence and the well-known “strip” lemma.

All proofs are immediate corollaries of general results on relations.

```
definition preds: relation term ≝ star … pred.

lemma preds_refl: reflexive … preds.

lemma preds_step_sn: ∀M1,M. M1 ⤇M →∀M2. M ⤇∗ M2 →M1 ⤇∗ M2.

lemma preds_step_dx: ∀M1,M. M1 ⤇∗ M →∀M2. M ⤇M2 →M1 ⤇∗ M2.

lemma preds_step_rc: ∀M1,M2. M1 ⤇M2 →M1 ⤇∗ M2.
```

lemma preds_compatible_abst: compatible_abst preds.

lemma preds_compatible_sn: compatible_sn preds.

lemma preds_compatible_dx: compatible_dx preds.

lemma preds_compatible_appl: compatible_appl preds.

lemma preds_lift: liftable preds.

lemma preds_inv_lift: deliftable_sn preds.

lemma preds_dsubst_dx: dsubstable_dx preds.

lemma preds_dsubst: dsubstable preds.

theorem preds_trans: transitive … preds.

lemma preds_strip: ∀M0,M1. M0 ⤇∗ M1 →∀M2. M0 ⤇M2 →
∃∃M. M1 ⤇M & M2 ⤇∗ M.

theorem preds_conf: confluent … preds.

4.6 Labeled Sequential Computation

Labeled sequential computation (`lsreds`) from M to N along the redexes pointed by the elements of s, is denoted here by $M \mapsto^*[s]\ N$. This notion is defined as the reflexive and transitive closure of labeled sequential reduction and comes with standard properties including stability under relocation and substitution.

Most proofs are immediate corollaries of general results on relations.

definition lsreds: ptrl → relation term $\overset{\text{def}}{=}$ lstar … lsred.

lemma lsreds_refl: reflexive … (lsreds (◇)).

lemma lsreds_step_sn: ∀p,M1,M. M1 ↦[p] M →∀s,M2. M ↦∗[s] M2 →M1 ↦∗[p::s] M2.

lemma lsreds_step_dx: ∀s,M1,M. M1 ↦∗[s] M →
∀p,M2. M ↦[p] M2 →M1 ↦∗[s@p::◇] M2.

lemma lsreds_step_rc: ∀p,M1,M2. M1 ↦[p] M2 →M1 ↦∗[p::◇] M2.

lemma lsreds_inv_nil: ∀s,M1,M2. M1 ↦∗[s] M2 →◇= s → M1 = M2.

lemma lsreds_inv_cons: ∀s,M1,M2. M1 ↦∗[s] M2 →∀q,r. q::r = s →
∃∃M. M1 ↦[q] M & M ↦∗[r] M2.

lemma lsreds_inv_step_rc: ∀p,M1,M2. M1 ↦∗[p::◇] M2 →M1 ↦[p] M2.

```
lemma lsreds_inv_pos: ∀s,M1,M2. M1 ↦∗[s] M2 →0 < |s| →
                        ∃∃p,r,M. p::r = s & M1 ↦[p] M & M ↦∗[r] M2.

lemma lsred_compatible_rc: ho_compatible_rc lsreds.

lemma lsreds_compatible_sn: ho_compatible_sn lsreds.

lemma lsreds_compatible_dx: ho_compatible_dx lsreds.

lemma lsreds_lift: ∀s. liftable (lsreds s).

lemma lsreds_inv_lift: ∀s. deliftable_sn (lsreds s).

lemma lsreds_dsubst: ∀s. dsubstable_dx (lsreds s).

theorem lsreds_mono: ∀s. singlevalued ... (lsreds s).

theorem lsreds_trans: ltransitive ... lsreds.

lemma lsreds_compatible_appl: ∀r,B1,B2. B1 ↦∗[r] B2 →∀s,A1,A2. A1 ↦∗[s] A2 →
                                @B1.A1 ↦∗[(sn:::r)@dx:::s] @B2.A2.

lemma lsreds_compatible_beta: ∀r,B1,B2. B1 ↦∗[r] B2 →∀s,A1,A2. A1 ↦∗[s] A2 →
                                @B1.λ.A1 ↦∗[(sn:::r)@(dx:::src::s)@◇::◇] [↙B2] A2.
```

The equivalence between labeled sequential reduction and parallel reduction, in that $M \Mapsto^* N$ iff $\exists s.\ M \mapsto^*[s]\ N$, takes to the Confluence Theorem `lsreds_conf`.

The first three proofs are by induction on the structure of the first premise.

```
theorem lsreds_preds: ∀s,M1,M2. M1 ↦∗[s] M2 →M1 ⤇∗ M2.

lemma pred_lsreds: ∀M1,M2. M1 ⤇M2 →∃r. M1 ↦∗[r] M2.

theorem preds_lsreds: ∀M1,M2. M1 ⤇∗ M2 →∃r. M1 ↦∗[r] M2.

theorem lsreds_conf: ∀s1,M0,M1. M0 ↦∗[s1] M1 →∀s2,M2. M0 ↦∗[s2] M2 →
                       ∃∃r1,r2,M. M1 ↦∗[r1] M & M2 ↦∗[r2] M.
```

Weak head computations play a fundamental role in the proof of the standardization theorem (Subsection 4.7), but cannot be characterized in terms of degree-based pointers. Therefore Kashima must define a dedicated relation for them, that he names “hap” in [Kas00], and that comes with its own theory. On the contrary, our path-based pointers allow to avoid this device since we easily characterize a weak head computation form M to N as $M \mapsto^*[s]\ N$ under the assumption `is_whd` s.

We want to stress that our Lemma `in_whd_ind` (Subsection 3.5) allows us to reason about weak head computations without generating more proof cases than Kashima does reasoning about his “hap” computation.

4.7 Decomposed Standard Computation

This section is devoted to the proof of the standardization theorem using an inductively defined decomposition of a standard computation from M to N as weak head computation followed by an inner computation. To this aim, Kashima defines a computation, denoted here by $M \circledS \mapsto^* N$, that he names "st" in [Kas00].

Our version of `st` comes with standard properties that include three inversion lemmas. Transitivity is proved after standardization rather than directly.

```
inductive st: relation term ≝
| st_vref : ∀s,M,i. is_whd s → M ↦∗[s] #i → st M (#i)
| st_abst : ∀s,M,A1,A2. is_whd s → M ↦∗[s] λ.A1 → st A1 A2 → st M (λ.A2)
| st_appl : ∀s,M,B1,B2,A1,A2. is_whd s →M ↦∗[s] @B1.A1 →
            st B1 B2 → st A1 A2 → st M (@B2.A2)
.

lemma st_inv_lref: ∀M,N. M Ⓢ↦∗ N →∀j. #j = N →
                   ∃∃s. is_whd s & M ↦∗[s] #j.

lemma st_inv_abst: ∀M,N. M Ⓢ↦∗ N →∀C2. λ.C2 = N →
                   ∃∃s,C1. is_whd s & M ↦∗[s] λ.C1 & C1 Ⓢ↦∗ C2.

lemma st_inv_appl: ∀M,N. M Ⓢ↦∗ N →∀D2,C2. @D2.C2 = N →
                   ∃∃s,D1,C1. is_whd s & M ↦∗[s] @D1.C1 &
                              D1 Ⓢ↦∗ D2 & C1 Ⓢ↦∗ C2.
lemma st_refl: reflexive … st.

lemma st_step_sn: ∀N1,N2. N1 Ⓢ↦∗ N2 →
                  ∀s,M. is_whd s → M ↦∗[s] N1 →M Ⓢ↦∗ N2.

lemma st_step_rc: ∀s,M1,M2. is_whd s →M1 ↦∗[s] M2 →M1 Ⓢ↦∗ M2.

lemma st_lift: liftable st.

lemma st_inv_lift: deliftable_sn st.

lemma st_dsubst: dsubstable st.
```

The Standardization Theorem `lsreds_standard` is proved in two steps. Firstly we prove that $M \mapsto^*[s]\ N$ implies $M \circledS \mapsto^* N$ (Lemma `st_lsreds`) by induction on s. Lemma `st_step_sx` (Lemma 3.6 of [Kas00]) is the inductive case of the proof. Secondly we prove that $M \circledS \mapsto^* N$ implies $M \mapsto^*[r]\ N$ such that `is_standard` r (Lemma `st_inv_lsreds_is_standard`) by induction on the structure of the premise.

We show the main proofs in full. The reader may note that the first proof case of Lemma `st_step_sx` is Lemma 3.5 of [Kas00].

```
lemma st_step_dx: ∀p,M,M2. M ↦[p] M2 →∀M1. M1 Ⓢ↦∗ M →M1 Ⓢ↦∗ M2.
#p #M #M2 #H elim H −p −M −M2
[ #B #A #M1 #H
  elim (st_inv_appl ... H ???) −H [4: // |2,3: skip ]
  #s #B1 #M #Hs #HM1 #HB1 #H
  elim (st_inv_abst ... H ??) −H [3: // |2: skip ] #r #A1 #Hr #HM #HA1
  lapply (lsreds_trans ... HM1 ...(dx ::: r) (@B1.λ.A1) ?) /2 width=1/ −M #HM1
  lapply (lsreds_step_dx ... HM1 (◇) ([↙B1]A1) ?) −HM1 // #HM1
  @(st_step_sn ... HM1) /2 width=1/ /4 width=1/
| #p #A #A2 #_ #IHA2 #M1 #H
  elim (st_inv_abst ... H ??) −H [3: // |2: skip ] /3 width=5/
| #p #B #B2 #A #_ #IHB2 #M1 #H
  elim (st_inv_appl ... H ???) −H [4: // |2,3: skip ] /3 width=7/
| #p #B #A #A2 #_ #IHA2 #M1 #H
  elim (st_inv_appl ... H ???) −H [4: // |2,3: skip ] /3 width=7/
]
qed−.

lemma st_lsreds: ∀s,M1,M2. M1 ↦∗[s] M2 →M1 Ⓢ↦∗ M2.

lemma st_inv_lsreds_is_standard: ∀M,N. M Ⓢ↦∗ N →
                                   ∃∃r. M ↦∗[r] N & is_standard r.
#M #N #H elim H −M −N
[ #s #M #i #Hs #HM
  lapply (is_whd_is_standard ... Hs) −Hs /2 width=3/
| #s #M #A1 #A2 #Hs #HM #_ ∗ #r #HA12 #Hr
  lapply (lsreds_trans ... HM (rc:::r) (λ.A2) ?) /2 width=1/ −A1 #HM
  @(ex2_intro ... HM) −M −A2 /3 width=1/
| #s #M #B1 #B2 #A1 #A2 #Hs #HM #_ #_
  ∗ #rb #HB12 #Hrb ∗ #ra #HA12 #Hra
  lapply (lsreds_trans ... HM (dx:::ra) (@B1.A2) ?) /2 width=1/ −A1 #HM
  lapply (lsreds_trans ... HM (sn:::rb) (@B2.A2) ?) /2 width=1/ −B1 #HM
  @(ex2_intro ... HM) −M −B2 −A2 >associative_append /3 width=1/
]
qed−.

theorem st_trans: transitive ... st.

theorem lsreds_standard: ∀s,M,N. M ↦∗[s] N →∃∃r. M ↦∗[r] N & is_standard r.
```

Finally, we add Theorem `lsreds_lsred_swap`, which states that weak head reduction steps can be taken in front of a computation. The inductive case of the proof, that we show in full, is very similar to Lemma 4.1 of [Kas00] stating that head reduction steps can be taken in front of a computation. Theorem `lsreds_lsred_swap` is the starting point for proving that a term has a weak head normal form iff every computation starting from it contains finitely many weak head reduction steps.

```
lemma st_lsred_swap: ∀p. in_whd p →∀N1,N2. N1 ↦[p] N2 →∀M1. M1 ⓢ↦∗ N1 →
                        ∃∃q,M2. in_whd q & M1 ↦[q] M2 & M2 ⓢ↦∗ N2.
#p #H @(in_whd_ind ...H) −p
[ #N1 #N2 #H1 #M1 #H2
  elim ( lsred_inv_nil  ... H1 ?) −H1 // #D #C #HN1 #HN2
  elim (st_inv_appl  ... H2 ... HN1) −N1 #s1 #D1 #N #Hs1 #HM1 #HD1 #H
  elim (st_inv_abst  ... H ??) −H [3: // |2: skip ] #s2 #C1 #Hs2 #HN #HC1
  lapply (lsreds_trans  ... HM1 ...(dx ::: s2)  (@D1.λ.C1) ?) /2 width=1/ −N #HM1
  lapply (lsreds_step_dx  ... HM1 (◇) ([↙D1]C1) ?) −HM1 // #HM1
  elim (lsreds_inv_pos  ... HM1 ?) −HM1
   [2: >length_append normalize in ⊢(??(??%)); // ]
  #q #r #M #Hq #HM1 #HM
  lapply (rewrite_r  ?? is_whd ... Hq) −Hq /4 width=1/ −s1 −s2 ∗ #Hq #Hr
  @(ex3_2_intro ... HM1) −M1 // −q
  @(st_step_sn  ... HM) /2 width=1/
| #p #_ #IHp #N1 #N2 #H1 #M1 #H2
  elim (lsred_inv_dx  ... H1 ??) −H1 [3: // |2: skip ]
  #D #C1 #C2 #HC12 #HN1 #HN2
  elim (st_inv_appl  ... H2 ... HN1) −N1 #s #B #A1 #Hs #HM1 #HBD #HAC1
  elim (IHp ... HC12 ...HAC1) −p −C1 #p #C1 #Hp #HAC1 #HC12
  lapply (lsreds_step_dx  ... HM1 (dx::p) (@B.C1) ?) −HM1 /2 width=1/ −A1 #HM1
  elim (lsreds_inv_pos  ... HM1 ?) −HM1
   [2: >length_append normalize in ⊢(??(??%)); // ]
  #q #r #M #Hq #HM1 #HM
  lapply (rewrite_r  ?? is_whd ... Hq) −Hq /4 width=1/ −p −s ∗ #Hq #Hr
  @(ex3_2_intro ... HM1) −M1 // −q /2 width=7/
]
qed−.

theorem lsreds_lsred_swap: ∀s,M1,N1. M1 ↦∗[s] N1 →
                              ∀p,N2. in_whd p →  N1 ↦[p] N2 →
                              ∃∃q,r,M2. in_whd q & M1 ↦[q] M2 &
                                         M2 ↦∗[r] N2 & is_standard (q::r).
```

In the proof of Lemma `st_lsred_swap` the reader should note two invocations of the tactic "lapply (`rewrite_r` ?? `is_whd` ...`Hq`)", which is the procedural counterpart of the declarative construction "cut (`is_whd` (q::`r`)) [>`Hq`]".

5. CONCLUSIONS AND FUTURE WORK

In this article we reported on a formalization of pure λ-calculus for the Matita interactive theorem prover [ARST11], which contains the proofs of two relevant results in reduction theory: the Confluence Theorem `lsreds_conf` of Subsection 4.6 and the Standardization Theorem `lsreds_standard` of Subsection 4.7.

Our aim was to provide what appears to be the first formalization of Kashima's standardization proof [Kas00] for a Logical Framework. On the other hand, confluence is a due result we provide for completeness following Tait's and Matin-Löf's proof based on the so-called "diamond" property of parallel reduction steps (Subsection 4.4) leading to confluence of parallel computations (Subsection 4.5), that has been formalized many times already (see for instance [Pfe92]).

Our main contribution was to adopt path-based pointers (Subsection 3.5) to locate the redexes of λ-terms. In our view this tool has two major benefits over Kashima's degree-based pointers. Firstly, we note that one step of arbitrary sequential reduction is defined by four clauses [Bar85], and we can modify this definition including a pointer to the contracted redex without increasing the number of clauses (Subsection 4.3). Secondly, we can characterize the weak head computations with our predicate `is_whd` of Subsection 3.6. So Kashima's "hap" relation is avoided. Such a characterization is impossible with Kashima's pointers.

On the other hand, our pointers can be ordered (see Subsection 4.1) as well as Kashima's ones in such a way that ordered sequences of pointers to redexes should correspond to standard computations. We plan to solve this conjecture by testing our definition of a standard computation against the definition based on residuals.

We also plan to prove the results on leftmost computations mentioned in [Kas00], such as the leftmost reduction theorem and the quasi-leftmost reduction theorem.

In this respect we note that Kashima's pointers allow a clean characterization of a leftmost reduction step as one for which the pointer to the contracted redex is 1.

We expect to obtain a characterization as well by saying that p is leftmost in M when is minimal in M, in that $M \mapsto [q]\ N$ implies $p \leq q$ for every q and N.

Finally we plan to consider η-reduction following Kashima in [Kas01].

The source files of our formalization are on Matita's Web site[4], while the corresponding proof objects are in the HELM directory[5] <cic:/matita/lambda/>.

References

[APS+03] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical Knowledge Management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1):27–46, May 2003.

[ARST11] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita Interactive Theorem Prover. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction (CADE-2011)*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69, Berlin, Germany, 2011. Springer.

[Bar85] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, The Netherlands, November 1985.

[Bar93] H.P. Barendregt. Lambda Calculi with Types. *Osborne Handbooks of Logic in Computer Science*, 2:117–309, 1993.

[Coq12] Coq development team. *The Coq Proof Assistant Reference Manual: release 8.4*. INRIA, Orsay, France, August 2012.

[CP90] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of the International Conference on Computer Logic (Colog '88)*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66, Berlin, Germany, 1990. Springer.

[4]<http://matita.cs.unibo.it/library.shtml>.
[5]<http://helm.cs.unibo.it/library.html>.

[dB94] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Selected Papers on Automath*, pages 375–388. North-Holland Pub. Co., Amsterdam, The Netherlands, 1994.

[Gui06] F. Guidi. lambdadelta_1. Formal specification with the proof assistant Coq 7.3.1, November 2006. Available at the $\lambda\delta$ Web site: <http://lambdadelta.info/>.

[Gui09] F. Guidi. The Formal System $\lambda\delta$. *Transactions on Computational Logic*, 11(1):5:1–5:37, November 2009.

[Gui10] F. Guidi. Procedural Representation of CIC Proof Terms. *Journal of Automated Reasoning*, 44(1-2):53–78, February 2010. Special Issue on Programming Languages and Mechanized Mathematics Systems.

[Hue94] G. Huet. Residual Theory in λ-calculus: A Formal Development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[Kas00] R. Kashima. A Proof of the Standardization Theorem in λ-Calculus. Technical Report RRMCS C-145, Tokyo Institute of Technology, Tokyo, Japan, August 2000.

[Kas01] R. Kashima. On the Standardization Theorem for $\lambda\beta\eta$-Calculus. International Workshop on Rewriting in Proof and Computation (RPC'01), October 2001.

[KN96] F. Kamareddine and R.P. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155(1):85–109, 1996.

[MP99] J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3):373–409, November 1999.

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming languages.* Series in Computer Science. Prentice-Hall International, Hemel Hempstead, UK, May 1987.

[Pfe92] F. Pfenning. A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework. Technical Report CMU-CS-92-186, Carneige-Mellon University, Pittsburgh, PA, USA, September 1992.

[RP04] S. Ronchi Della Rocca and L. Paolini. *The Parametric Lambda Calculus.* Texts in Theoretical Computer Science. Springer, Berlin, Germany, November 2004.

[Tak95] M. Takahashi. Parallel Reductions in λ-Calculus. *Information and Computation*, 118(1):120–127, April 1995.

[Xi99] H. Xi. Upper Bounds for Standardizations and An Application. *Journal of Symbolic Logic*, 64(1):291–303, 1999.

ACKNOWLEDGMENTS

My sincere thanks go to professor A. Asperti for his valuable comments and suggestions, which greatly improved this text.

I dedicate this work on the beautiful subject of λ-calculus to (in alphabetical order without repetitions): Ada, Angela, Annalisa, Camilla, Chiara, Elena, Élodie, Fiorella, Francesca, Gloria, Ida, Isabel, Laura, Loredana, Lucia, Luki, Mariella, Marta, Patrizia, Rosanna, Silvia, and to the other friends of SPaCe, Bologna.

“[Beautiful] is what you love.”
Sappho, Book I.