A Formalization of the Theory of Objects in Isabelle/HOL

Article · January 2006

Source: OAI

CITATIONS
0

READS
152

2 authors:

Ludovic Henrio
French National Centre for Scientific Research
150 PUBLICATIONS **1,306** CITATIONS

SEE PROFILE

Florian Kammüller
Middlesex University, UK
119 PUBLICATIONS **974** CITATIONS

SEE PROFILE

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Formalization of the Theory of Objects in Isabelle/HOL

Ludovic HENRIO — Florian Kammüller

N° ????

Decembre 2006

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

A Formalization of the Theory of Objects in Isabelle/HOL

Ludovic HENRIO*, Florian Kammüller†

Thème COM — Systèmes communicants
Projet Oasis

Rapport de recherche n° ???? — Decembre 2006 — 17 pages

Abstract: We present a formalization of Abadi's and Cardelli's theory of objects in the interactive theorem prover Isabelle/HOL. In particular, we present (a) a formal model of objects and its operational semantics based on DeBruijn indices (b) a parallel reduction relation for objects (c) the proof of confluence for the theory of objects reusing Nipkow's HOL-framework for the lambda calculus.

Key-words: sigma-calculus, semantics, confluence, formal model

* CNRS – I3S – INRIA, Sophia-Antipolis
† Technische Universität Berlin

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Un modèle pour la théorie des objets en Isabelle/HOL

Résumé : Ce document présente un modèle formel pour a théorie des objets d'Abadi et Cardelli sous Isabelle/HOL. Nous présentons ici un modèle formel pour les objets et leur sémantique basé sur les indices de DeBruijn ; une operation de réduction parallèle pour les objets ; et une preuve de confluence pour le sigma-calcul réutilisant la contribution de Nipkow pour la confluence du lambda-calcul.

Mots-clés : sigma-calcul, sémantique, confluence, modèle formel

1 Introduction

The Theory of Objects [1] defines the ς-calculus for the abstract and precise characterization of object oriented languages. The ς-calculus is a computation model for object orientated programming in the same way as the λ-calculus models functional programming.

Ever since its creation, the ς-calculus has evolved in many ways. First, the authors of [1] already provide a wide range of different extensions for the basic ς-calculus (e.g., [2] and [3]), summarized in the book [1]. The Theory of Objects has also been adopted by many as the *lingua franca* for the theory of object oriented programming and has been taken as a basis for further experimentation and development. For example, Gordon and Hankin extended the ς-calculus towards the paradigm of parallel programming [12]. More recently, the ς-calculus has been incorporated into the ASP calculus that is a theoretical basis for distributed objects [9], and also into higher-level flavors like aspect-orientation [16].

The objective of this paper is to provide a sound foundation and formalization of the ς-calculus. We also expect this work to ground further formalizations of extensions and concepts relying on the ς-calculus, and to impact significantly on the mechanized proofs of the correctness of such extensions. We are particularly interested in the design of distributed versions of the ς-calculus, and as such, in proving confluence first for the ς-calculus in order to lift the mechanization to parallelized object calculi. Indeed, in the presence of distributed objects, confluence is recognized as a particularly interesting question.

For those projects, and more generally aiming at a wide use of a mechanized theory of objects, we present here a formalization and confluence proof of the untyped ς-calculus. It strongly uses of a framework for confluence in Isabelle/HOL [17], and an earlier attempt on the formalization of the ς-calculus [11].

A first idea could consist in proving confluence in the ς-calculus by relying on its translation into the λ-calculus [2] which is confluent. However, objects are lost in the translation into the λ-calculus, which prevents us from concluding about the confluence in the object world (no function has been defined yet for bringing back a lambda term into an object world – which is a priori impossible). Moreover, a mechanized model adapted to objects allow us to aim at several crucial properties on objects, like typing, confluence of concurrent object languages, etc. We detail some of these perspectives in Section 5.

In this paper we first introduce Isabelle/HOL and the ς-calculus in Section 2 to provide sufficient technical detail for the understanding of the exposition. Then, in Section 3 we present the model as expressed in the input language of Isabelle/HOL. Section 4 introduces confluence proofs, as provided by the framework of Tobias Nipkow [17], and then presents the derivation of confluence for the ς-calculus. The Isabelle/HOL mechanization is available at one of the authors' web page [15].

2 Preliminaries

In this section we introduce Isabelle/HOL and the functional ς-calculus; both with regard to the elements that are relevant for the understanding of the remainder of the paper.

2.1 Isabelle/HOL

The interactive theorem prover Isabelle has foremost been constructed as a generic tool to provide a framework for the creation of specialized theorem provers for various application logics. However, besides Isabelle/ZF, an embedding of Zermel-Fraenkel set theory it is the instantiation to Higher Order Logic (HOL), called Isabelle/HOL, that is nowadays most widely used. In particular for computer science applications, where typing comes in naturally, HOL is well-suited as it provides a logic with types. The deductive engine of Isabelle is its meta logic, itself a fragment of HOL: it contains just implication and universal quantification as junctors. There is no existential quantifier, no negation, no disjunction. Conjunction is mimicked using nested implication. The following meta-logical formula is an example illustrating the universal quantification with `!!`, higher order variables `P` and `Q`, and implication $\Longrightarrow$.

```
!! P Q x. [| P x; Q x |] ⟹ P x
```

The square brackets `[| |]` serve as pseudo-conjunction: they are just an abbreviation for nested implication, i.e the above is equivalent to

```
!! P Q x. P x ⟹ (Q x  ⟹ P x)
```

where the round brackets could even be omitted as the meta-level implication is right-associative.

Moreover, the object logic HOL contains all the classical logic constructors, as $\longrightarrow$ for implication, $\forall$ and $\exists$ for quantification, $\wedge$ for conjunction and $\vee$ for disjunction.

In computer science applications we often reason about rather simple domains, like discrete structures, finite sets, partial recursive or primitive recursive functions. For such specialized domains, Isabelle/HOL offers also specialized support for formalization and proof. These specializations are internally resolved and mapped to the principles of conservative extension.

2.1.1 HOL Example

Let us focus on the data type of lists which is a good introduction for the constructions and capabilities of Isabelle/HOL used in this article. Lists can be defined in Isabelle/HOL using the `datatype` definition package. A `datatype` definition strongly looks like the corresponding ML version.

```
datatype α list =
   Nil  ("[]")
 | Cons α (α list)  (infixr "#" 65)
```

Using the polymorphism of the type system of Isabelle/HOL the above definition introduces the type `list` over an arbitrary type of elements. The datatype definition introduces a constructor `Nil` for the empty list and a constructor `Cons` that, given an element of type α, and a list of α elements, constructs a new list. The code in brackets behind the constructors

Let $o \equiv [l_j = \varsigma(x_j)b_j]^{j\in 1..n}$ (l_j distinct).

o	is an object with method names l_j and methods $\varsigma(x_j)b_j$	
$o.l_j$	$\rightarrow_\beta b_j\{x_j \leftarrow o\}$	$(j \in 1..n)$ selection / method call
$o.l_j := \varsigma(x)b$	$\rightarrow_\beta [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i^{i\in(1..n)-\{j\}}]$	$(j \in 1..n)$ update / override

declares the pretty printing syntax enabling the use of `[]` for the empty list and `x # l` for a constructed list.

Among the internally generated rules for a datatype specification there are induction rules for recursive types like the above and injectivity rules for the constructors.

Functions over a datatype may be defined as primitive recursive functions. As an illustrative example consider the function that appends two list to form a new one. First, we declare this function as a constant in a theory.

```
consts
  append :: [α list, α list] ⇒ α list (infixr "@" 65)
```

Next, the semantics of this function is given by the two classical equations below. The names before the colon `:` are optional rule names for later reference in proofs.

```
primrec
  append_Nil: [] @ l = l
  append_Cons: (x # l1) @ l2 = x # (l1 @ l2)
```

The primitive recursion schema constraints the way recursion can be defined to ensure that primrec functions are actually primitive recursive. This schema provides automatic and optimized tactics. Hence, for example, if a term needs to be transformed using equational rewriting given by a primitive recursive function definition, this is performed fully automatically by Isabelle/HOL.

2.2 Functional ς-Calculus

The Theory of Objects consists in various ς-calculi that are aimed to be as "simple and fruitful as λ-calculi" [2]. Rather than using the λ-calculus to encode objects and their behaviour in a way that is overly complicated, the ς-calculus takes objects as primitive.

The kernel calculus that we model in this paper includes *object definition*, *method invocation*, an *method override*. An object consists of a set of labeled fields. A field can be a method or a simple field. A method is a function with one formal parameter that represents *self*, i.e., the object in which the method is contained. A field is just a degenerate method not using its self parameter. Therefore selection of a field or invocation (call) of a method of an object are identical. Similarly method override and field update are also interchangeable. We quote next the so-called primitive semantics of objects [2]. For a gentler introduction we refer to the following section where we introduce the ς-calculus step by step in Isabelle/HOL.

This definition implicitly relies on the following syntax:

$$\begin{array}{lll} a, b ::= & [l_j = \varsigma(x_j)b_j]^{j\in 1..n} & \text{object definition} \\ & \mid a.l_j & (j \in 1..n) \text{ method call} \\ & \mid a.l_j := \varsigma(x)b & (j \in 1..n) \text{ update} \end{array}$$

3 Isabelle/HOL Model

In this section we introduce the formalization of the ς-calculus with DeBruijn indices [6]. We then show how substitution is formalized on the DeBruijn object terms and how it works technically based on lifting. Finally, we define the reduction relation $\rightarrow_\beta$ and show some first proof results concerning the transitive, reflexive closure $\rightarrow_\beta^*$ of $\rightarrow_\beta$.

The formalization of the ς-calculus by Ould Ehmety [11] in Isabelle/ZF, seems to have followed the earlier formalization of the λ-calculus in Isabelle/HOL [17]. It also uses DeBruijn indices. Although, Ould Ehmety's formalization of ς-terms, substitution, and the reduction relation has been performed in Isabelle/ZF, they are close enough to Nipkow's λ-formalization in HOL and can be used here. The main reason for such a similarity is that ZF supports datatype definitions in a very similar style as HOL. However, we deviate from Ould Ehmety in that we choose lists instead of maps for representing objects. Concerning the proofs, in the formalization of Ould Ehmety, they were unfinished. But they seem to be directed at typing results. Therefore, for confluence we had to start from scratch.

3.1 Object Terms using DeBruijn Indices

DeBruijn indices are very useful for implementations of calculi with abstraction as they abstract from variable names. A variable is replaced by a natural number that represents the distance — in terms of nesting depth — of this variable to its binder. Thereby terms contain only numbers no variables; α-conversion becomes obsolete. This is a considerable advantage from a pragmatic point of view as α-conversion is a difficult problem. Also for mechanical proofs α-conversion is a known hard problem having already triggered recent research activities [20].

DeBruijn indices are best explained by an example. Consider the following term on the left side in the well known form of λ-calculus with variables and its equivalent on the right side with DeBruijn indices.[1]

$$\lambda x.\lambda y.(\lambda z.\, x\, z)y = Abs(Abs(Abs(Var\,2)\$(Var\,0))(Var\,0))$$

Note that, different variables may be represented by the same number, e.g., z and x both are $Var\,0$. DeBruijn indices relieves one from having to deal with α-conversion: for example both $\lambda x.x$ and its α-equivalent $\lambda y.y$ are represented by $Abs(Var0)$. The downside of DeBruijn indices is that substitution, crucial for the definition of application, is rather

[1] We use here the constructors Var, Abs, and \$ for variables, abstractions, and application as in Isabelle/HOL.

complicated to define: a term has to be "lifted", i.e. his "variables" have to be increased by one, when it moves into the scope of an abstraction in the process of substitution. We will encounter this definition for ς in the next subsection.

In the ς-calculus, abstraction is used to represent the self of an object as a parameter in a method $\varsigma(x)b$ that is replaced by the current enclosing object when this method is called. This abstraction will be represented by DeBruijn indices. Hence, variables are represented as natural numbers. The type `dB` of ς-terms in Isabelle/HOL is given by the following datatype declaration where `Label` is just a type synonym for `nat`, the type of natural numbers.

```
datatype dB =
    Var nat
  | Obj dB list
  | Call dB Label
  | Upd dB Label dB
```

The constructor `Var` builds-up a new term `dB` from a `nat` representing the DeBruijn index of the variable. The constructor `Obj` takes a `list` of fields or methods as parameters; even a method $\varsigma(x)b$ having a formal parameter x is a simple `dB` term. The constructor for invocation `Call` selects a field given by a label in a `dB` term representing an object. Field update (method override) `Upd` replaces a labelled field in an object by another value, i.e., a `dB` term. This informal semantics will be formally encoded by the definition of the reduction relation $\rightarrow_\beta$ in Section 3.3. In order to define the reduction we need to define substitution on these DeBruijn terms. The fact that we use a list to represent the indexed set of labelled fields in an object will be discussed at the end of this section in 3.4.

3.2 Substitution

As DeBruijn indices discard the use of formal parameters, substitution has to be performed by adapting the numbers representing variables when a term is moved between different layers of the nested scopes of abstraction. This movement occurs precisely when a variable has to be substituted by a term containing a free variable inside the scope of an abstraction. Therefore the notion of substitution is chained with the notion of lifting. We declare the following two constants in Isabelle/HOL.

```
subst ::     [dB, dB, nat] ⇒ dB  ("_[_'/_]" [300, 0, 0] 300)
lift ::      [dB, nat] ⇒ dB
```

Because of the declared mixfix syntax, we can write `t[s/n]` to express that in a term `t` the variable represented by `n` shall be replaced by `s`. Before defining the semantics of substitution we need to define the lifting of a term. A lifting carries a parameter `n` representing the *cut* between free and bound variable numbers in the term that shall be lifted. The operation `lift` is defined by the following set of primitive recursive equations describing the effect of lifting over the various cases of object terms.

```
liftVar:    lift (Var i) k = (if i < k then Var i else Var (i + 1))
```

```
liftObj:    lift (Obj f) k = (Obj (map (λ x. lift x (k + 1)) f))
liftCall:   lift (Call a l) k = Call (lift a k) l
liftUpd:    lift (Upd a l b) k = Upd (lift a k) l (lift b (k + 1))
```

A variable is only lifted when it is free, i.e. when its representing number is greater or equal the "cut" parameter. The "cut" parameter is increased in the recursive call when an abstraction scope is entered. This is the case when the lift function enters inside a method in an object, and when a field is updated by a method. Note that we increase only on the right side of an update because the left side will always be an object seen as a reference whereas the right side is a method.[2]

Substitution can now be defined in terms of lift as follows.

```
subst_Var:   Var i [s/k] =
              if k < i then Var (i - 1) else if i = k then s else Var i
subst_Obj:   Obj f [s/k] = Obj (map (λ x. x[(lift s 0)/(k+1)]) f)
subst_Call:  Call a l [s/k] = Call (a [s/k]) l
subst_Upd:   Upd a l b [s/k] = Upd (a [s/k]) l (b [lift s 0 / k+1])
```

The idea is that a term `s` is lifted if it is substituted inside an abstraction scope, i.e. inside an object and at the right side of an update. The lifting is always initiated with "cut" parameter `0` as initially the outermost variable free when entering a scope.[3] The decrementation in the equation for `Var` in cases of free variables greater than the "cut" parameter is not necessary for substitution itself but is needed for β-reduction, i.e., the relation `beta` that we present next.

3.3 Reduction Relation

Once substitution is defined the reduction relation can easily be specified. We first declare a relation `beta` (sometimes denoted by $\rightarrow_\beta$ in the following) as a set of pairs of object terms and then use a translation to define the convenient syntax $s \rightarrow_\beta t$ that we are used to.

```
consts
  beta :: (dB × dB) set
translations
  s →β t == (s, t) ∈ beta
  s →β* t == (s, t) ∈ beta^*
```

The relation $\rightarrow_\beta$ is now defined by an inductive definition. Given a set in Isabelle/HOL an inductive definition consists of a set of rules — adhering to certain well-formedness rules —defining the contents of the set in an inductive style. The definition of the set is then implicitly given by the smallest set closed under those rules. As a consequence, induction schemes can be automatically provided by Isabelle/HOL.

[2]For clarity of the exposition we use here the `map` function in `liftObj`. In reality this is rejected by Isabelle/HOL as it violates the primitive recursion scheme. An individual function `map_lift` has to be defined.

[3]Concerning the `map` in `subst` the same problem occurs as mentioned in the previous footnote 3.2.

```
inductive beta
intros
  beta: l < length f ⟹ Call (Obj f) l →β (f!l)[(Obj f)/0]
  upd : Upd (Obj f) l a →β  Obj (f [l := a])
  sel : s →β t ⟹ Call s l →β Call t l
  updL: s →β t ⟹ Upd s l u →β Upd t l u
  updR: s →β t ⟹ Upd u l s →β Upd u l t
  obj : s →β t ⟹ Obj (f [l := s]) →β Obj (f [l := t])
```

The central and most interesting rule of the reduction is the first rule `beta` that initiates an evaluation of $o.l$ by replacing the lth field, say $\sigma(x)b$, of the object o for the formal self parameter x. The other rules define the reduction relation $\rightarrow_\beta$ to be a congruence, i.e., we can reduce terms inside contexts. In the concrete syntax we profit from the natural style that is defined for lists in Isabelle/HOL: for example to extract the nth element of an object `Obj f` we can write `f ! n`. Similarly the update by `x` is `Obj (f [n := x])`.

For the investigation of the reduction relation, in particular for confluence, we need to investigate the transitive, reflexive closure $\rightarrow_\beta^*$ of $\rightarrow_\beta$. Isabelle/HOL provides sufficient support in its theory database for reasoning about relations. For example, for any relation `r` of type `(`$\alpha\times\alpha$`)set` the reflexive, transitive closure may be constructed as `r^*`; corresponding theorems and induction scheme are provided.

3.3.1 Congruence Rules for $\rightarrow_\beta^*$

For the transitive reflexive closure $\rightarrow_\beta^*$ of $\rightarrow_\beta$ the following congruence rules can be derived.

```
s →β* s' ⟹ Call s l →β* Call s' l
s →β* s' ⟹ Upd s l u →β* Upd s' l u
s →β* s' ⟹ Upd u l s →β* Upd u l s'
[| u →β* u'; s →β* s' |] ⟹ Upd u l s →β* Upd u' l s'
s →β* s' ⟹ Obj(f [l := s]) →β* Obj(f [l := s'])
```

The last rule is a direct transposition of the rule `obj` of $\rightarrow_\beta$ to its transitive closure $\rightarrow_\beta^*$. For the use in the confluence proofs however the following derived rule is more suitable.

```
[| n < length f; f ! n →β* x |] ⟹ Obj (f) →β* Obj (f[n := x])
```

3.4 Extensions for Typing

Evidently our HOL model of objects is not quite adequate with respect to one point: we use lists for the fields of an object where the original Theory of Objects prescribes a sequence of labels mapping to terms. The reasons for this deviation are pragmatic. The type system of classical HOL as encoded in Isabelle/HOL is such that all function are total. Hence, the type usually used for maps is the `Map`-type that mimics a partial function type by the total function type $\alpha\Rightarrow$`(`β `option)` where β `option` is the lifting of an arbitrary type β given by the following datatype.

```
datatype α option = None | Some α
```

The `option` type together with pattern matching enables a smooth treatment of partiality sufficient for many applications.

In the earlier ZF-formalization of ς-calculus [11], it is this option type that has been used to model the map contained in an object. Unfortunately, there is no natural and nicely embedded version of *f*inite maps available. It appears that in most proofs, eventually, the finiteness is not necessary to reach the results. Unfortunately, in our case it, is a necessary prerequisite (see for example the lemma of Section 4.4).

Furthermore, lists are well supported, their syntax is very close to maps, and finally using list update, we implicitly respect the "domain" of a map, i.e. an update out of bounds is ignored as described in the following theorem.

```
!! i. length xs ≤ i ⟹ xs[i:=x] = xs
```

Moreover, there are several inductions on lists available: structural list induction, simultaneous structural induction, structural induction in reverse form, i.e., over `l @ [x]`, and an induction over the length of lists. Clearly, we could have defined a finite type of maps, or a class of finite types and assume maps in that class. In any case, we would have had to construct this infrastructure first before being able to begin with the formalization of the ς-calculus.

On the other hand, the inadequacy of our model is not irreversible. In fact we can add types later on by extending an object `Obj f` with an additional map from list indices to labels. This works in principle as depicted in Figure 1.

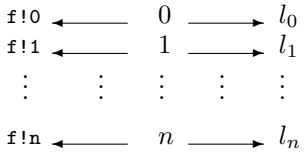

Figure 1: Extension of object by map to labels for typing

As the list selection `λi. l ! i` represents a function, and as the map from indices to labels is injective, we can invert it and associate to each label a unique term. In [1], types of objects are defined by their labels, and we can easily provide an extension for typing by integrating labels in our model as explained above. The proof of confluence will not be influenced by such a change. From a general point of view, dealing with natural numbers instead of labels makes the handling of the formalization simpler.

4 Confluence Proof

4.1 Nipkow's Framework

Tobias Nipkow provides in [17] a framework for the proof of Church-Rosser properties in Isabelle/HOL. By "framework" we mean that his formalization is in large parts reusable. Although he formalizes only the classical λ-calculus and its operational semantics, the proof of confluence is mainly conducted on a generic level using the polymorphic relation type $(\alpha \times \alpha)$`set`. Therefore, it constitutes a reusable proof enabling the reduction of a confluence proof to central lemmata as shown in this section.

Nipkow follows in his formalization the classical way of proving Church-Rosser as explained in Barendregt's book [4][Chapter 3]. Apparently, it is also this proof method, originated by Tait and Martin-Löf, that is used by Abadi and Cardelli for proving Church-Rosser [2]. Nipkow moreover formalizes an alternative approach of the so-called *complete developments* due to Takahashi shorter and more elegant on paper. For the mechanical proof there is no gain because the classical proof is solved almost automatically by Isabelle.

We give an outline of the main properties of the framework for confluence proofs. The property `square` is a predicate over four relations describing confluence of a relation in its most general from.

```
square :: [(α × α)set, (α × α)set, (α × α)set, (α × α)set] ⇒ bool
 square R S T U ==
 ∀ x y. (x, y)∈ R ⟶ (∀ z. (x, z)∈ S ⟶ (∃ u. (y, u)∈ T ∧ (z, u)∈ U))
```

The square predicate is used as a primitive in proofs as it enables reasoning similar to graphical arguments where we express confluence as a square as depicted in the diagram of Figure 2.

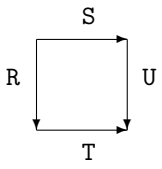

Figure 2: The square predicate

In general, and also in our case, we want to prove the square with just one relation (the transitive, reflexive closure of the reduction relation) at each edge. Therefore, `commute` reduces the square to just two relations and `diamond` to one. Finally confluence is defined as a square over just one, the closure of a relation.

```
commute :: [(α × α)set, (α × α)set] ⇒ bool
commute R S == square R S S R
```

```
diamond :: ((α × α)set ⇒ bool)  diamond R == commute R R

confluent :: (α × α)set ⇒ bool  confluent R == diamond (R^*)
```

The original Church-Rosser property describes that any two terms that are connected by the relation or its inverse have a common reduct.

```
Church_Rosser :: (α × α) set ⇒ bool
"Church_Rosser R ==
  ∀ x y. (x, y) × (R ∪ R^-1)^* ⟶ (∃ z. (x, z) ∈ R^* ∧ (y, z) ∧ R^*)"
```

The following general theorem represents the classical reduction of the Church-Rosser property to confluence, i.e., diamond property of the closure of the reduction relation.

```
Church_Rosser_confluent: "Church_Rosser R = confluent R"
```

The following theorem provides a further possible reduction of confluence of a relation `T` to proving the diamond property of a relation `R` in between `T` and its reflexive transitive closure.

```
diamond_to_confluence: [| diamond R; T ⊆ R; R ⊆ T^* |] ⟹ confluent T
```

The classical trick used also in the application of the framework to the λ-calculus is to use a so-called parallel reduction for `R` for which the diamond property is true and simple to prove. Indeed, in general, the original reduction relation does not verify `diamond T`, and proving `diamond T^*` is very difficult without using a parallel reduction. Then we only have to show the inclusion of the parallel reduction relation in between the original reduction relation `T` and its transitive, reflexive closure.

4.2 Parallel Reduction

In order to reuse the full extent of Nipkow's framework we have to define a parallel reduction relation for the ς-calculus. In general, a parallel reduction relation is a relation similar to the original reduction relation, but able to reduce several subterm of the original term: it applies reduction at several possible places at the same time. Hence, the main difficulty is to find such a relation that accumulates somehow the original relation — and define this relation in such a way that it matches the provisos of Theorem `diamond_to_confluence`, i.e., lies in between the original reduction `beta` and its transitive, reflexive closure `beta^*`.

The parallel reduction relation for the ς-calculus that we use is very similar to its equivalent in the λ-calculus: it simply applies itself recursively at all possible reduction places, and includes the reflexive relation. It is defined as follows:

```
syntax
  par_beta :: ([dB, dB] => bool)  (infixl "⇒β" 50)
translations
  s ⇒β t == (s, t) ∈ par_beta
```

```
inductive par_beta
  intros
    var:  Var n ⇒β Var n
    obj:  [| length s = length s'; ∀ l < length s. s!l ⇒β s'!l |]
          ⟹ Obj s ⇒β Obj s'
    upd:  [| s ⇒β s'; t ⇒β t' |] ⟹ Upd s l t ⇒β Upd s' l t'
    upd': [| Obj s ⇒β Obj s'; t ⇒β t' |]
          ⟹ (Upd (Obj s) l t)  ⇒β (Obj (s' [l := t']))
    sel:  s ⇒β t ⟹ Call s l ⇒β Call t l
    beta: [| Obj f ⇒β Obj f'; l < length f' |]
          ⟹ Call (Obj f) l ⇒β (f' ! l)[(Obj f')/0]
```

4.3 Inclusion Lemmata and Diamond Property of `par_beta`

The framework of Nipkow provides the general structure of the proof of confluence for a reduction relation on terms. Hence, in order to show Church-Rosser one has to show confluence. Furthermore, showing confluence is reduced to showing that the parallel reduction `par_beta` is between `beta` and `beta^*` (`beta` ⊆ `par_beta` ⊆ `beta^*`) and that the diamond property holds for `par_beta`.

Now we cannot get much more (for free) from the framework. However, we can try to follow the outline of the proofs of these properties in the case of the λ-calculus. In Nipkow's proof all three lemmata are solved almost automatically by Isabelle's classical reasoner, but, in the case of the ς-calculus, we need to interact more and to prove some cases manually.

The proof of `beta` ⊆ `par_beta` is performed using induction and Isabelle's classical reasoner. It needs decisively more guidance than the original proof.

The other inclusion `par_beta` ⊆ `beta^*` is in principle comparable. However, it revealed a lemma that we needed to solve separately (see Section 4.4).

The diamond property `diamond par_beta` finally is rather long and technical in our case. There are a considerable number of combinations between the different constructors leading to numerous cases in the case analysis. Like Nipkow we start the global proof by unfolding the definitions of `diamond`, `commute`, and `square`, and applying `par_beta` induction on the unfolded goal. In contrast to Nipkow, where the rest is done automatically by one application of the classical reasoner, we need to guide the prover on the remaining subgoals. A typical subgoal is the following:

```
[| length s = length s';                                    (1)
   ∀ l<length s. s!l ⇒β s'!l ⟶
   (∀ z. s!l ⇒β z ⟶ (∃ u. s'!l ⇒β u ∧ z ⇒β u))
|] ⟹ ∀ z. Obj s ⇒β z ⟶ ∃ u. Obj s' ⇒β u ∧ z ⇒β u)
```

This goal basically means that the diamond property can be lifted to objects, provided it is verified (by recurrence) on all the fields of the object. To solve this goal we use an inversion lemma for objects:

```
[| Obj s ⇒β z |] ⟹ ∃ lz. length s = length lz ∧ z = Obj lz
```

The application of this lemma gives a witness `z = Obj lz` with `Obj s` $\Rightarrow_\beta$`Obj z`. Unfortunately the proviso for the right lower half of the diamond square in the goal (1) `(∀z. s!l` $\Rightarrow_\beta$`z` $\longrightarrow$`(∃u. s' ! l` $\Rightarrow_\beta$`u ∧z` $\Rightarrow_\beta$`u))` is too fine grained. We need another technical lemma that transforms this proviso into the existence of a list of elements.

```
(∃ lu. length lu = length s ∧
          (∀ l < length s. s'!l ⇒β lu!l ∧ lz!l ⇒β lu!l))
```

Using the witness list `lu` we can then insert `Obj lu` as the existential witness that represents the lower right corner of the diamond square (`u` in the goal (1)).

For the remaining two subgoals `Obj s'` $\Rightarrow_\beta$`Obj lu`, and `Obj lz` $\Rightarrow_\beta$`Obj lu` we simply apply twice the object reduction lemma that we present in the next section, and has, in fact, already been derived for the proof of `par_beta` $\subseteq$ `beta^*`.

4.4 Object Reduction Lemma

In the proof of `par_beta` $\subseteq$ `beta^*` and the diamond property for `par_beta` we encounter the following subgoal:

```
[| length f = length g; ∀ l < length f. f!l →β* g!l |]                (2)
⟹ Obj f →β* Obj g
```

This goal trivially occurs when reduction for objects can be applied; such a reduction reduces simultaneously all fields of an object. Using the recurrence hypothesis, we can infer that each of the field can be obtained by `beta^*`, and we want to prove that this can be lifted to the level of the object (roughly: $\rightarrow^*_\beta \rightarrow^*_\beta \ldots \rightarrow^*_\beta = \rightarrow^*_\beta$).

Although seemingly obvious it is not trivial to prove. We first derive the following lemma that describes the witness of a list that keeps record of all steps in a $\rightarrow_\beta$ step by step transformation from the field map f to the field map g. This transformation is described graphically in Figure 3.

```
lemma rtrancl_beta_obj_lem:
[| length f = length g; ∀ l < length f. f!l →β* g!l |] ⟹
  ∀ k ≤ length f.
    (∃ ob. length ob = (k + 1) ∧
      (∀ obi. obi mem ob ⟶ length obi = length f) ∧
      (ob ! 0 = f) ∧ (Obj (ob ! 0) →β* Obj (ob ! k)) ∧
      (take k (ob ! k) = take k g) ∧
      (drop k (ob ! k) = drop k f))
```

The functions `take` and `drop` are predefined list operators. Given a natural number `n` and a list `l` the application `take n l` returns the list containing the `n` first elements of `l`; `drop n l` returns the rest of `l` when the first `n` are dropped. Using the existence of a list `ob` for each `n` $\leq$ `length f` we can prove the initial subgoal (2) using the lemma `rtrancl_beta_obj_lem` instantiated with `length f`. Having the existence of `ob`, we then only need to infer that its last element is equal to `g`.

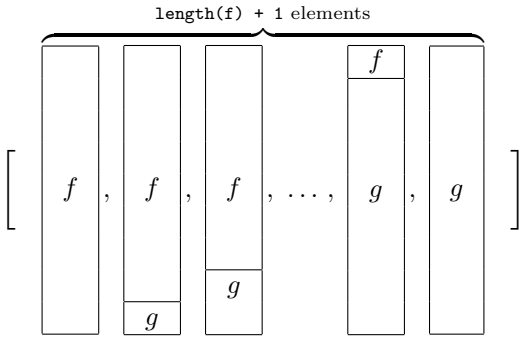

Figure 3: List of stepwise transformations

4.5 Confluence

The proof of the confluence property for the ς-calculus is, thanks to Nipkow's framework, simply achieved by proving the theorem `diamond_to_confluence` appropriately instantiated.

```
[| diamond par_beta; beta ⊆ par_beta; beta ⊆ beta^* |]⟹ confluent beta
```

The provisos of this main theorem, i.e., `diamond par_beta`, `beta` $\subseteq$ `par_beta`, and `par_beta` $\subseteq$ `beta^*` are the lemmata described in the penultimate section and just have to be plugged in. Thereby we have shown that the reduction relation $\rightarrow_\beta$ for the ς-calculus as defined here is Church-Rosser. This corresponds to the result in the original paper [2][Theorem 2.1-1].

5 Conclusion, Impact and Perspectives

In this paper we have presented the formalization of the ς-calculus in Isabelle/HOL using a de Bruijn notation. We have formalized the syntax and its operational semantics and proved confluence. We did profit from an earlier model in Isabelle/ZF and the mechanization of the λ-calculus with DeBruijn indices. The latter could be used as a framework for our proofs. We used a pragmatic representation of lists to contain the fields of an object. Although differing from the original Theory of Objects we argue that no harm is done. Besides a mechanical verification of the ς-calculus the value of our contribution is as a basis for future mechanical models of object oriented languages.

In the presence of distribution, confluence is a particularly interesting question. Therefore we are interested in proving confluence first for the ς-calculus in order to lift the mechanization to distributed object calculi. In practice, this work should first lead to a mechanized version of the ASP calculus [9, 10]. This calculus extends the imperative ς-calculus [1] by adding distribution primitives. It mainly relies on the aggregation of objects into so-called activities, and asynchronous method calls between such activities, *futures* acting as promised replies associated to such calls. The ASP-calculus is the theoretical basis for active objects

as implemented in the ProActive library. A first step in order to build a mechanized version of ASP could consist in investigating a simpler *functional* version of ASP, for this we plan to rely on the framework presented in this paper.

A further motivation for the mechanization of the ς-calculus is given by the project Ascot [14] for the mechanically supported analysis of aspect-oriented languages. We intend to use the formalization of the ς-calculus as presented in this paper to model and examine type safety of a core aspect calculus.

Another classical extension of this work consists in bringing all the typing theory presented in [1] into the Isabelle/HOL framework for ς-calculus in order to mechanize the proofs of subject reduction and type properties exhibited ten years ago by Abadi and Cardelli.

Finally, a lot of theoretical results have been the objective of previous research on object calculi, e.g., [19, 8, 7] for concurrency, [5] for mobility, [13] for a bisimilarity relation, etc. Those results generally rely on a calculus very close to the ς-calculus (and sometimes on the ς-calculus itself). Thanks to the mechanized aspect of our model, we expect our framework to be useful in order to verify and perhaps improve the properties shown in those various contexts.

Acknowledgment: We would like to thank Larry Paulson for providing us the formalization of the ς-calculus in Isabelle/ZF written by Ould Ehmety.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, New York, 1996.

[2] Martín Abadi and Luca Cardelli. *A Theory of Primitive Objects.* DEC Research Labs, TR, 1995.

[3] Martín Abadi and Luca Cardelli. An imperative object calculus. TAPSOFT'95: Theory and Practice of Software Development. Volume 915 of LNCS, Springer, 1995.

[4] Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

[5] Sébastien Briais and Uwe Nestmann. Mobile objects "must" move safely. In *Formal Methods for Open Object-Based Distributed Systems IV – Proceedings of FMOODS'2002, University of Twente, the Netherlands.* Kluwer Academic Publishers, 2002.

[6] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, **34**:381–392, 1972.

[7] Paolo Di Blasio and Kathleen Fisher. A calculus for concurrent objects. In *International Conference on Concurrency Theory*, 1996.

[8] Luca Cardelli. A language with distributed scope. In *Conference Record of the 22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'95).*

[9] D. Caromel and L. Henrio. *A Theory of Distributed Objects.* Springer-Verlag, 2005.

[10] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, 2004.

[11] Sidi Ould Ehmety *Theory of objects in Isabelle/ZF.* Unpublished theory files, 1999.

[12] Andrew D. Gordon and Paul D. Hankin. *A concurrent object calculus: reduction and typing.* In Proceedings HLCL'98, volume 16. Elsevier ENTCS, 1998.

[13] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96).*

[14] S. Jähnichen and F. Kammüller. *Ascot: Formal, mechanical foundation of aspect-oriented and collaboration-based languages.* Project with the German Research Foundation (DFG), 2006.

[15] F. Kammüller. Author's web-page. `http://swt.cs.tu-berlin.de/~flokam`, 2006.

[16] Jay Ligatti, David Walker and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming.* Springer 2006.

[17] Tobias Nipkow. More Church Rosser Proofs. *Journal of Automated Reasoning.* **26**:51–66, 2001.

[18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, Springer LNCS, **828**, 1994.

[19] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Proceedings of Theory and Practice of Parallel Programming (TPPP'94), Sendai, Japan*, LNCS. Springer-Verlag, 1995.

[20] Christian Urban et al. Nominal Methods Group. Web-page at `http://www4.in.tum.de/~urbanc/Nominal/`. Project funded by the German Research Foundation (DFG) within the Emmy-Noether Programme, 2006.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399