*Electronic Notes in Theoretical Computer Science 58 No. 1 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume58.htm*](http://www.elsevier.nl/locate/entcs/volume58.html)*l 17 pages*

*A First-Order Syntax for the -Calculus in* Isabelle/HOL using Permutations

*Christine Rockl 1 LAMP { DI { EPFL*

*Ecole Polytechnique F ed erale de Lausanne INR Ecublens, CH-1015 Lausanne, Switzerland*

*Abstract*

*A formalized theory of alpha-conversion for the -calculus in Isabelle/HOL is pre- sented. Following a recent proposal by Gabbay and Pitts, substitutions are modelled in terms of permutations, and alpha-equivalence is de ned over all but nitely many names. In contrast to the work by Gabbay and Pitts, however, standard universal and existential quanti cation are used instead of introducing a new binder.*

*Further, a classi cation of the various approaches to formalizing languages with binders is presented. Strengths and weaknesses are pointed out, and suggestions for possible applications are made.*

# *1 Introduction*

*The -calculus is a model of higher-order concurrent programming languages* [[3](#_bookmark19)[,18,22,31](#_bookmark41)], and as such is particularly characterized by its binders, input,

*def*

*ax:P , and restriction, ( x)P . As an example, consider the processes P =*

*!az:zx :zx :P 0 and C d=ef ( y)a y:y b:y c:C0. Process P is a procedure* [*2*](#_bookmark1) *that each* time it is called along channel a and transmitted two arguments, launches an instantiation of its body P 0. Process C, on the other hand, models a client creating a new channel y over which it then transmits its parameters b and c to P . Assuming that z does not occur in P 0 and y does not occur in C0, the following communication can be established between the two processes

*1 2*

*1 Email:* [*christine.roeckl@epfl.c*](mailto:christine.roeckl@epfl.ch)*h*

*2 The de nition of procedures is a typical application of replication !P , intuitively describing an unlimited number of copies of P . Often replication is used in an input-bounded form,*

*!az:P , where a can be understood as the name of the procedure.*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license*](http://creativecommons.org/licenses/by-nc-nd/3.0/)*.*

*(modulo strong bisimilarity ):*

*0 0*

*P j C ! P j ( y)(yx1:yx2:P j y b:y c:C ) C calls P along a,*

*0 0 0*

*! P j ( y)(yx2:P fb=x1 gj y c:C ) C transmits b to P ,*

*0 0 0*

*! P j P fb=x1 ; c=x2gj C C transmits c to P .*

*This simple mechanism of creating and transmitting new names is the source* of the expressive power of the -calculus, but makes reasoning extremely in- tricate, necessitating an application of interactive theorem-proving.

*The rst question to be decided in formalizations of languages with binders,* like the -calculus, is which syntax to build on. There exist three general approaches, each of which gives rise to a number of variations: (1) rst-order syntax in a deep embedding, (2) higher-order syntax in a deep embedding, and (3) higher-order syntax in a shallow embedding. Note that rst-order syntax always yields a deep embedding. Traditionally, deep embeddings are considered to be well-understood and are therefore often applied in syntax analysis and meta-theoretical justi cations. Reasoning about concrete sys- tems of larger size, on the other hand, entails cumbersome de nitions as well as a di using administration of bound variables. Therefore, shallow embed- dings have become more and more popular, motivated both by a growing the- oretical basis and the fact that they free the user from struggling with bound variables, the latter being particularly essential in reasoning about concrete terms (programs or processes). On the other hand, reasoning about syntax in a shallow embedding either requires a logical framework that is suÆciently weak yet incorporates non-standard axioms ([16,17] presents such a framework implemented for the -calculus in Coq), or else a heavy machinery eliminating exotic terms and mimicking the missing structural induction principles by rule induction over well-formedness predicates ([28,29] presents such a framework for the -calculus in Isabelle/HOL). Note that the actual choice of a speci c syntactic framework heavily relies on the application one has in mind.

*In this paper, we study a formalization of the -calculus in Isabelle/HOL [27,25]* using a rst-order syntax 3 . It is part of a larger project aiming at theorem- prover support for reasoning within and about the -calculus and related languages. The main motivation for the work at hand was to see how con- veniently the permutation model can be applied to a formalization of the - calculus using Isabelle/HOL (so far, Gabbay and Pitts have used Isabelle/ZF as a framework to justify their approach and apply it to the -calculus, see [8] for details), and to relate it to approaches investigated previously. The propo- sition of Gabbay and Pitts [8,9] is based on two ideas: (1) model substitutions in terms of permutations and (2) reason about all but nitely many instan- tiations. As a slight modi cation, we do not introduce a new operator for

*3 The proof-scripts are available at* [*http://lampwww.epfl.ch/~roeckl/Pi/.*](http://lampwww.epfl.ch/~roeckl/Pi/) *Some ex- tracts can also be found in the appendix.*

*idea (2) but implement it in terms of universal and existential quanti cation.* Note that idea (2) is independent of idea (1); it can be applied together with other notions of substitution alike.

*A second aim of this paper is to generally classify frameworks for languages* with binders, based on our experience with the -calculus. That is, we give characterizations of the diverse techniques, discussing their strengths and weaknesses, and specify areas in which they can typically be applied. Al- though originally obtained for the -calculus in Isabelle/HOL, these results should be equally valid for formalizations of other languages with binders in comparable frameworks.

*Overview*

*In Section* [*2,*](#_bookmark3) *we describe the basic features of Isabelle/HOL. In Section* [*3*](#_bookmark4)*,* we introduce the syntax of the -calculus and model substitutions by permuta- tions, in Isabelle/HOL. In Section [4,](#_bookmark7) we derive a theory of -equivalence using all but nitely many instantiations. Section [5](#_bookmark14) discusses the diverse approaches providing guidelines for applications. Section [6](#_bookmark16) concludes the paper.

# *2 Isabelle/HOL*

*We use the general-purpose theorem-prover Isabelle* [*[27*](#_bookmark37)*], based on higher-* order intuitionistic logic, and formalize the -calculus in its instantiation HOL for higher-order logic [[25].](#_bookmark35) Proofs in Isabelle are based on uni cation, and are usually conducted in a backward-resolution style: the user formulates the goal he/she intends to prove, and then|in interaction with Isabelle| continuously reduces it to simpler subgoals until all of the subgoals have been accepted by the tool. Upon this, the goal can be stored in the theorem- database of Isabelle/HOL to be applicable in further proofs. The prover o ers various tactics, most of them applying to single subgoals. The basic resolution tactic resolve\_tac, for instance, allows the user to instantiate a theorem from Isabelle's database so that its conclusion can be applied to transform a current subgoal into instantiations of its premises. Besides these classical tactics, Isabelle o ers simpli cation tactics based on algebraic transformations. Powerful automatic tactics apply the basic tactics to prove given subgoals according to di erent heuristics. The automatic solver auto\_tac, for example, combines classical reasoning and simpli cation over sets of rules that can be modi ed by the user. The tactic

*(auto\_tac (claset() addIs [nren\_trans], simpset() delsimps [nren\_def]))*

*for example, adds the introduction rule nren\_trans to the set of classical* rules, and removes the rewrite rule nren\_def from the set of simpli cation rules. When adding de nitions and theorems, however, the user has to take care not to provoke unprovable subgoals or in nite loops.

*In Isabelle/HOL, the user can de ne recursive datatypes and inductive* sets. The prover then automatically computes rules for induction and case- injection. Note that all these techniques have been fully formalized and veri-

*ed in Isabelle/HOL, that is, they are a conservative generic extension [2,26].*

*De nitions and proofs are usually given on an object-level (HOL), but can* employ meta-level (Isabelle) functions. This gives rise to shallow embeddings as opposed to deep embeddings fully residing on the object-level [2].

# *3 The* [*Syntax*](#_bookmark34) *of the -Calculus*

*The -calculus [23,24] is the mobile counterpart of CCS, reducing its prede-* cessor to basics by identifying the sort of messages and channels, referring to both as names. This gives processes the possibility to declare new names and send them to other processes, thus create private communication lines with them. It is this syntactic simplicity, from which the -calculus draws the power to model mobile communication systems [23] and higher-order lan- guages [3,18,22,31].

*Names*

*Consider an at least countably in nite set of names, ranged over by a; b;::: ;* x; y;:: :. In our formalization, we have not chosen a particular type, but axiomatically accept every type that is at least countably in nite. We have adopted this approach already in [28,29], leaving the possibility for speci c instantiations, such as with naturals or reals. Naturals, for instance, can increase automation in certain cases, because fresh names can be computed by determining the maximal name in the involved processes and choosing the successor.

*Processes*

*Implementing communications as basic primitive, the calculus possesses* pre xes ::= j a b j ax, which are silent, that is, invisible, output, and input pre xes, respectively. Processes are then built from inaction, pre x, restric- tion, choice, parallel composition, matching, mismatching, and replication:

*P ::= 0 j :P j ( y)P j P + Q j P j Q j [a = b]P j [a 6= b]P j !P:*

*The actual syntax may vary from one application to another. Often, mis-* matching is omitted due to its semantic misbehaviour in various cases, or output pre xes may stand for themselves only, as in asynchronous -calculi. We formalize the full syntax to be comparable with previous works [17,28,29].

*Binders*

*The -calculus has two binders, input pre x and restriction, for which the* notion of boundedness is slightly distinct, however. A name x bound in an

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *fn (0)*  *fn ( :P )* | *=*  *=* | *;*  *fn (P )* | *bn (0)*  *bn ( :P )* | *=*  *=* | *;*  *bn (P )* |
| *fn (a b:P )* | *=* | *fa; bg[ fn (P )* | *bn (a b:P )* | *=* | *bn (P )* |
| *fn (ax:P )* | *=* | *fag[ (fn (P ) fxg)* | *bn (ax:P )* | *=* | *fxg[ bn (P )* |
| *fn (( x)P )* | *=* | *fn (P ) fxg* | *bn (( x)P )* | *=* | *fxg[ bn (P )* |
| *fn (P + Q)* | *=* | *fn (P ) [ fn (Q)* | *bn (P + Q)* | *=* | *bn (P ) [ bn (Q)* |
| *fn (P j Q)* | *=* | *fn (P ) [ fn (Q)* | *bn (P j Q)* | *=* | *bn (P ) [ bn (Q)* |
| *fn ([a = b]P )* | *=* | *fa; bg[ fn (P )* | *bn ([a = b]P )* | *=* | *fa; bg[ bn (P )* |
| *fn ([a 6= b]P )* | *=* | *fa; bg[ fn (P )* | *bn ([a 6= b]P )* | *=* | *fa; bg[ bn (P )* |
| *fn (!P )* | *=* | *fn (P )* | *bn (!P )* | *=* | *bn (P )* |

*Table 1*

*Names of processes. Free and bound names are computed by primitively recursive functions. The names of a process are n (P ) = fn (P ) [ bn (P ).*

*def*

*input ax:P can be considered as a place-holder for a name b to be received in*

*a communication ax:P j a b:Q ! P fb=xgj Q later on. A name y bound in a*

*restriction ( y)P , on the other hand, rather represents a private channel, that* is, a name that is only known to P and is inaccessible to all processes running in the environment. On a more nominal level, this exclusiveness could also be explained as y being distinct from any name currently in use.

*Free and bound names*

*In order to compute the free and bound names of a process P , we use* standard primitively recursive functions fn (P ) and bn (P ). The names of a process, n (P ), are then simply the union of its free and bound names. For instance, fn (ax:P ) = fag[ (fn (P ) fxg) and bn (ax:P ) = fxg[ bn (P ). See Table 1 for a complete overview. Note that we are able to compute the bound names only because we are working with a deep embedding. In a shallow embedding, bound names are meta-variables, inaccessible on the object-level.

*Substitution*

*The use of rst-order syntax entails the need for substitution, in order to*

*(1) de ne -equivalence and -conversion and to (2) instantiate terms in a*

*-reduction. Focusing on the rst area of application, we can use permuta-* tions to implement substitution. Table 2 gives the corresponding de nitions for the -calculus. Note that instantiations require a non-injective notion of substitution.

*The approach has two elegant properties: (1) in contrast to standard sub-* stitutions, permutations are completely symmetric, and (2) permutations can deal with free and bound names alike, owing to their bijectivity. The permu- tation model allows us to derive the following results without any diÆculty:

|  |  |  |
| --- | --- | --- |
| *(0)fx $ yg*  *( :P )fx $ yg* | *=*  *=* | *0*  *:P fx $ yg* |
| *(a b:P )fx $ yg* | *=* | *afx $ ygbfx $ yg:P fx $ yg* |
| *(ab:P )fx $ yg* | *=* | *afx $ ygbfx $ yg:P fx $ yg* |
| *(( b)P )fx $ yg* | *=* | *( bfx $ yg)P fx $ yg* |
| *(P + Q)fx $ yg* | *=* | *P fx $ yg + Qfx $ yg* |
| *(P j Q)fx $ yg* | *=* | *P fx $ ygj Qfx $ yg* |
| *([a = b]P )fx $ yg* | *=* | *[afx $ yg = bfx $ yg]P fx $ yg* |
| *([a 6= b]P )fx $ yg* | *=* | *[afx $ yg =6 bfx $ yg]P fx $ yg* |
| *(!P )fx $ yg* | *=* | *!P fx $ yg* |

*Table 2*

*Substitution. Permuting on names, afx $ yg = if a = x then y else if a = y then x else a, substitution can disregard whether a name is free or bound.*

*Lemma 3.1 For a process P and names a; b; x; y such that x; y 62 n (P ),*

*(i) P fa $ ag = P ,*

*(ii) P fa $ bg = P fb $ ag,*

*(iii) (P fx $ ag)fy $ xg = P fy $ ag.*

*Proof. The proofs are straightforward applications of Isabelle's automatic* tactics and, in item 3, structural induction. *2*

# *4 A Theory of* *Equivalence*

*In this section, we derive a theory of -equivalence following the second idea* of Gabbay and Pitts, which is to underspecify the requirements on names used in instantiations, simply referring to all but nitely many of them. This allows us to derive laws telling that it nally suÆces to instantiate continuations of binders with a single fresh name. Yet, in contrast to the approach proposed in [8,9], we do not introduce a new quanti er but specify it on two levels using universal and existential quanti cation. In particular, we rst introduce

*-equivalence with respect to a set F of \forbidden" names, in the de nition*

*of which we use universal quanti cation over all names not in F. Then we* de ne a notion independent of F which merely requires the existence of some

*nite F. Usually, one will choose the set of (free) names of the processes* that are to be compared. Yet, the underspeci cation of such a set allows us to derive transitivity of -equivalence, which is hard (or even impossible) for more speci c formulations.

*0 =F 0 nl*

*P =F P 0*

*:P =F :P 0*

*tau*

*P =F P 0*

*a b:P =F a b:P 0*

*out*

*8b 62 F: P fb $ xg =F[fbg P 0fb $ x0g ax:P =F ax0:P 0 in*

*P =F P 0 Q =F Q0*

*8b 62 F: P fb $ xg =F[fbg P 0fb $ x0g ( x)P =F ( x0 )P 0*

*P =F P 0 Q =F Q0*

*res*

*ch par*

*P + Q =F P 0 + Q0*

*P =F P 0*

*P j Q =F P 0 j Q0*

*P =F P 0 P =F P 0*

*[a = b]P =F [a = b]P 0 mt*

*[a 6= b]P =F [a 6= b]P 0*

*Table 3*

*mmt*

*!P =F !P 0*

*rep*

*-Equivalence wrt. F. We de ne -equivalence with respect to a set F of*

*\forbidden" names, that is, names that must not be used in instantiations.*

*Implementing -equivalence*

*Table 3 gives an overview of the introduction rules inductively de ning*

*-equivalence with respect to F. Intuitively, F speci es the (free) names of* the processes. Thus, when a fresh name is introduced by an instantiation, it has to be added to F. An equivalence statement ax:P =F ax0:P 0, for instance, can be derived from P fb $ xg =F[fbg P 0fb $ x0g. A basic result necessary in later proofs is that F can be augmented arbitrarily:

*Lemma 4.1 If P =F P 0, then P =F[F0 P 0.*

*Proof. By an easy induction on =F , solving all cases by means of the auto-* matic tactic auto\_tac (see also Appendix C). *2*

*De nition 4.2 [ -Equivalence] Two processes P and P 0 are* *equivalent,*

*written P = P 0, if there exists a nite F such that P =F P 0.*

*Results*

*The underspeci cation of F in De nition 4.1 allows us to derive transitivity* of -equivalence, as well as certain congruence results.

*Theorem 4.3 For processes P; P 0 and names x; y 62 n (P ) [ n (P 0),*

*(i) = is an equivalence.*

*(ii) P = P 0 implies P fy $ zg = P 0fy $ zg.*

*(iii) P fx $ bg = P 0fx $ b0g implies P fy $ bg = P 0fy $ b0g.*

*Proof. The results follow from similar results for =F and the fact that P =F*

*P 0 implies P =F[F0 P 0 for arbitrary F; F0 (transitivity). The equivalence*

*results are obtained easily using standard automatic tactics. The substitution* resulults are a bit harder to prove, using lists of permutations and induction over =F . *2*

*An immediate consequence of item 3 is that we do not have to consider all* instantiations with fresh names but only a single one in order to derive -

|  |  |  |
| --- | --- | --- |
| *(0)fx yg* | *=* | *0* |
| *( :P )fx yg* | *=* | *:P fx yg* |
| *(a b:P )fx yg* | *=* | *a b:P fx yg* |
| *(ab:P )fx yg* | *=* | *if b = y then ax:P fx $ yg else ab:P fx yg* |
| *(( b)P )fx yg* | *=* | *if b = y then ( x)P fx $ yg else ( b)P fx yg* |
| *(P + Q)fx yg* | *=* | *P fx yg + Qfx yg* |
| *(P j Q)fx yg* | *=* | *P fx ygj Qfx yg* |
| *([a = b]P )fx yg* | *=* | *[a = b]P fx yg* |
| *([a 6= b]P )fx yg* | *=* | *[a 6= b]P fx yg* |
| *(!P )fx yg* | *=* | *!P fx yg* |

*Table 4*

*-Conversion. The function P fx yg searches for the outermost bound occurrences of y replacing them by x and applying permutation to the continuation. Usually, a fresh x is chosen for that purpose.*

*equivalence of processes with binders. This yields the following characteriza-* tion:

*Theorem 4.4 For processes P; P 0; Q; Q0 and x 62 n (P ) [ n (P 0),*

*(i) 0 = 0,*

*(ii) P = P 0 implies :P = :P 0 and a b:P = a b:P 0 and [a = b]P = [a =* b]P 0 and [a 6= b]P = [a 6= b]P 0 and !P = !P 0,

*(iii) P = P 0 and Q = Q0 implies P + Q = P 0 + Q0 and P j Q = P 0 j Q0,*

*(iv) P fx $ bg = P 0fx $ b0g implies ab:P* [*=*](#_bookmark10) *ab0:P 0 and ( b)P = ( b0)P 0.* Proof. Follows as a corollary of Theorem 4.3. *2*

*A theory of* *conversion*

*In practice, a theory of -equivalence is often complemented by a notion* of -conversion. It can be speci ed by introducing fresh names for certain bound names, applying substitution underneath binders. Table 4 de nes a primitively recursive function implementing -conversion. For it, we can de- rive the standard laws characterizing -conversion, including that arbitrary bound names can be eliminated by replacing them with fresh ones.

*Theorem 4.5 For every process P and names a; b; x with x 62 n (P ),*

*(i) ax:P fx $ bg = ab:P , and*

*(ii) ( x)P fx $ bg = ( b)P ,*

*(iii) b 62 n (P fx bg),*

*(iv) P fx bg = P , and consequently,*

|  |  |  |  |
| --- | --- | --- | --- |
|  | *rst-order/deep* | *higher-order/deep* | *higher-order/shallow* |
| *adequacy* | *usually obvious* | *to be proved* | *to be proved* |
| *substitution* | *for whole lan-* *guage* | *for -calculus* | *not necessary* |
|  | *substitutions* | *deferred* | *for free* |
| *bound parameters* | *object-variables accessible* | *object-variables accessible* | *meta-variables inaccessible* |
| *induction* | *yes* | *no* | *no* |
| *exotic terms* | *no* | *no* | *possibly* |
| *application* | *meta-theory* *justify paper proofs* | *meta-theory*  *easy results about binders*  *-conversion* | *concrete exam- ples*  *easy results*  *about binders* |
| *-calculus* | *[1,12,13,20,28]* | *[10,11]* | *[5,17,21,28,29]* |

*Table 5*

*General classi cation. Combining rst-order and higher-order syntax with deep and shallow embeddings. First-order syntax always yields a deep embedding.*

*(v) there exists P 0 such that P = P 0 and b 62 n (P 0)*[*.*](#_bookmark12)

*Proof. Items 1 and 2 can be derived from Theorem 4.4 by applying transi-* tivity of permutation and re exivity of -equivalence. Items 3 and 4 follow by structural induction on P using items 1 and 2. Item 5 is a direct conse- quence. *2*

# *5 Classifying Formalizations*

*When it comes to formalizing a language, the rst question that naturally* arises is which syntax to choose. In this section, we present a classi cation of the approaches known up to date, based on (mostly own) practical experience. We describe the main features of the approaches, point out strengths and weaknesses with respect to formalizations, and try to give a guideline for possible areas of application. Sticking to the -calculus as an exemplaric language, we point to formalizations of it within the various schemes. In a second part of this section, we discuss in more detail rst-order formalizations; for discussions about higher-order formalizations, see, for instance, [11,17,29].

*General classi cation*

*There exist two ways of expressing binders in general-purpose theorem-* proving. Following a deep strategy, binders are formalized fully within the

*object-level using object-variables, for free and bound parameters alike. An* alternative way is to apply a shallow implementation strategy, de ning binders in terms of meta-level functions, thus representing bound names by meta- variables, whereas free parameters are further denoted by object-variables. These two implementation strategies can be combined with a rst-order or a higher-order syntactic description of the language. Applying rst-order syn- tax, of course, one always obtains a deep embedding, because it does not distinguish between binders and other operators on the syntactic level. Ta- ble 5 gives an overview of the three ways of formalization.

*(1) The classical way is to remain fully within the object-level of the prover,* giving a rst-order syntax in a deep embedding. It is usually close to the way languages are treated on paper, and is therefore traditionally applied in meta- theoretical reasoning. A major inconvenience with respect to both formaliza- tion and derivations is that substitutions and -equivalence have to be de ned explicitely. In particular, reasoning about concrete processes becomes rather cumbersome. Yet, in cases where one wants to reason about bound names|in a theory of -conversion, for example|the approach is indispensable.

*(2) A second line of research applies higher-order abstract syntax (HOAS), yet* still within a deep embedding. That is, a -calculus is formalized along the lines of approach (1) in order to provide a functional mechanism (a pseudo- meta-level) within which binders are then expressed. As a consequence, sub- stitutions and -equivalence have to be de ned only for the (small) underlying calculus and not for the whole (usually large) language itself. A de nite ad- vantage of the approach with respect to a shallow embedding is that it allows the user to choose an appropriate meta-logic. However, still both free and bound parameters are expressed by object-variables, which entails substitu- tions and makes the de nition of concrete terms (programs to be analysed) cumbersome.

*(3) A third line of research builds on HOAS in a shallow embedding, using the* functional mechanism of the theorem-provers to represent and deal with bound names. In this case, the user does not have to bother about -conversion and

*-reduction at all, but further looses access to the bound names. It is even the*

*case that free and bound names are not merely distinct but incomparable (on* the object-level, on which proofs are conducted) with the former being object- variables and the latter meta-variables. The approach is particularly useful to concrete processes, and enjoys increasing popularity also in meta-theoretical reasoning. On the other hand, the meta-levels provided by general-purpose theorem-provers are often so powerful that exotic terms can arise, making an axiomatisation of syntactic properties extremely delicate (see [14,16,17,29]).

*Concerning application, deep embeddings are rather suitable in meta-theory,* because they are (intuitively) close to reasoning on paper and naturally pro- vide structural induction. Shallow embeddings traditionally head for applica-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *straightforward* | *permutations* *[8,9]* | *PTS [19]* | *deBruijn [4]* |
| *method* | *simple renam- ing, substitu- tion on top* | *permutations* | *parameters versus vari-*  *ables, two*  *substitutions* | *nameless variables* |
|  | *-equivalence,*  *-conversion,*  *-reduction* | *- equivalence,*  *-conversion* | *- equivalence,*  *-conversion,*  *-reduction* | *- equivalence is identity,*  *-reduction* |
| *application* | *adequacy,*  *(syntax), semantics* | *(adequacy), syntax* | *syntax, semantics* | *semantics* |
| *-calculus* | *[1,20,28]* | *this paper* | *[12]* | *[13]* |

*Table 6*

*Formalizing rst-order syntax. From left to right, the schemes go from \close to de nitions on paper" to \more or less an implementation".*

*tions concerning concrete examples, that is, processes or programs. A general* problem of the HOAS approach in items (2) and (3) is that it does not easily provide structural induction, which makes syntax analysis diÆcult. However, with growing e ort in deriving induction and syntactic proof principles, HOAS is entering meta-theory as well. Induction can be incorporated in the logical framework [16] or mimicked by rule induction over well-formedness predi- cates [6,29]. For a general proof-framework, it might make sense to provide both kinds of syntaxes, with functions translating one syntax into the other and back.

*First-order syntax*

*To the best of our knowledge, four approaches have been studied in -* calculus encodings; see Table 6 for an overview. For other languages, further variations have been investigated (see, for instance, [7,15,30]).

*(1) The most straightforward way is to formalize textual substitution in terms* of simple rewrite rules, without taking care of name-capture. Such a de ni- tion can then be used to implement -conversion, so that a \proper" notion of substitution can take care of name-capture, usually applying normalization. A theory of -equivalence can be obtained along the same lines as done for approach (2) in this paper. However, the proofs usually require more inter- action, because of the asymmetry of the operator, with respect to free and bound parameters.

*(2) Recently, Gabbay and Pitts have proposed to use the FM-set model as a*

*basis for reasoning about languages with binders, modelling substitutions in* terms of permutations. This approach is characterized by a complete symme- try both of the renaming operation and of the treatment of free and bound parameters, which relies on the bijectivity of permutations. As demonstrated in [8,9] as well as in the preceeding sections of this paper, the permutation model allows for convenient de nitions of -conversion and -equivalence, as well as derivations of the respective theories. A practical drawback is that the injectivity of permutations forbids a formulation of -reduction, hence semantic analysis necessitates the introduction of a second notion of substi- tution along the lines of approach (1); note that for approach (3), a di erent datatype de nition is often convenient to prevent disjoint summation, and that both approaches (3) and (4) do not need a theory of -equivalence in the sense of (2).

*(3) As a compromise between an intuitive straightforward approach and tech-* nical feasibility, McKinna and Pollack propose the use of Pure Type Systems (PTS), based on the idea of explicitely distinguishing between free and bound atoms by means of distinct sorts, which they refer to as parameters and vari- ables. As a consequence, two notions of substitution have to be de ned, a plain textual version for parameters and a capture-avoiding notion for vari- ables. The resulting de nitions of substitution are surprisingly simple, yet they do not yield the symmetry properties inherent to the permutation model.

*(4) DeBruijn indices are regularly applied to implement functional mechanisms* on meta-levels of theorem-provers. Here parameters are replaced by numbers, and -equivalent terms are represented by one and the same implementation. Capture-avoiding instantiations are then expressed in terms of basic arith- metic operations that can be e ectively dealt with by the programming lan- guage used for the implementation. When formalizing languages with binders larger than the -calculus, the approach is hard to apply, because its technical orientation makes it intricate and prone to errors. Further, general-purpose theorem provers generally do not o er arithmetic operations as primitives, hence using DeBruijn indices never yields the degree of automation one would like to expect from it.

*It seems that in rst-order syntax with names, one will always have to formal-* ize two notions of substitution when taking semantic analysis into account. In syntactic analysis, one can do with a single notion of substitution following one of the rst three approaches. Following approach (1), one can require that the instantiated parameters be fresh, approach (2) o ers a single notion of substitution anyway, and approach (3) only necessitates substitution on variables. Approach (4) is only a restricted basis for syntax analysis, because it identi es -equivalent terms, and thus can only be applied in reasoning about -equivalence classes.

# *6 Conclusion*

*In this paper, we have pursued two goals: (1) examine meta-theoretical rea-* soning for the -calculus on the basis of two recent ideas of Gabbay and Pitts [8,9], and (2) classify syntactic frameworks for it and related languages.

*(1) We have formalized substitution in terms of permutations, and have in-* stantiated continuations of binders with all but nitely many names in the de nition of -equivalence. An appealing property of permutations is their symmetry, P fa $ bg = P fb $ ag, simplifying some proofs. Further, due to their bijectivity, permutations allow for an equal treatment of free and bound names, as well as for a convenient derivation of laws necessary to study - equivalence. This bijectivity yields, on the other hand, that they are incapable of describing -reduction even in name-passing calculi like the -calculus. As

*an example, consider the communication*

*a b:P j ax:bx:0*

*! P j (bx:0)fb=xg.*

*The result should reduce to P j bb:0, for which a non-injective substitution is* necessary, mapping both b and x to b.

*Concerning -equivalence, we have slightly modi ed the proposal of Gab-* bay and Pitts to deal with standard universal and existential quanti cation instead of having to introduce and develop a theory for a new quanti er. The main idea of the approach|underspecifying the names to be used in an instantiation|is retained, however, allowing for the use of single fresh names in instantiations. The formulation of -equivalence is independent of the use of permutations, and can hence be combined with other notions of substitution alike.

*(2) Languages with binders can be formalized in a deep or in a shallow embed-* ding. While the rst is traditionally considered closer to reasoning on paper, the latter is usually more amenable to concrete examples. The choice of a

*rst-order syntax is often motivated by its natural structural induction prin-*

*ciples. Higher-order syntax, on the other hand, is generally more convenient in* reasoning about binders, and currently there is some e ort to derive suitable principles for syntax analysis. Yet, the obtained results have to be adapted to the original syntax afterwards, if they are to justify proofs conducted on paper (returning from equivalence classes to plain terms).

# *References*

*[1] O. Ait-Mohamed. Pi-Calculus Theory in HOL. PhD thesis, Henry Poincarr e University, Nancy, 1996.*

*[2] S. Berghofer and M. Wenzel. Inductive datatypes in HOL|lessons learned in Formal-Logic Engineering. In Proc. TPHOL'99, volume 1690 of LNCS, pages 19{36, 1999.*

*[3] S. Dal Zilio and A. Gordon. Region analysis in a -calculus with groups. In*

*Proc. MFCS'00, volume 1893 of LNCS, pages 1{20. Springer, 2000.*

*[4] N. deBruijn. 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.*

*[5] J. Despeyroux. A higher-order speci cation of the -calculus. In Proc. TCS'00,* *volume 1872 of LNCS, pages 425{439. Springer, 2000.*

*[6] J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction* *in Coq. In Proc. LPAR'94, volume 822 of LNCS, pages 159{173. Springer, 1994.*

*[7] J. Ford and I. Mason. Establishing a general context lemma in PVS. In Proc.* *AWCL'01, 2001.*

*[8] M. Gabbay. A Theory of Inductive De nitions with* *Equivalence: Semantics,*

*Implementation, Programming Language. PhD thesis, Cambridge University,* *2000.*

*[9] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders.* *In Proc. LICS'99, volume 158, pages 214{224. IEEE, 1999.*

*[10] S. Gay. A framework for the formalisation of pi-calculus type systems in* *Isabelle/HOL. Technical report, University of Glasgow, 2000.*

*[11] A. Gordon and T. Melham. Five axioms of alpha-conversion. In Proc. TPHOL'96, volume 1125 of LNCS, pages 173{190. Springer, 1996.*

*[12] L. Henry-Gr eard. Proof of the subject reduction property for a pi-calculus in Coq. Technical Report RR-3698, INRIA, 1999.*

*[13] D. Hirschko . A full formalisation of -calculus theory in the calculus of constructions. In Proc. TPHOL'97, volume 1275 of LNCS, pages 153{169.* *Springer, 1997.*

*[14] M. Hofmann. Semantical analysis of higher-order abstract syntax. In Proc. LICS'99, volume 158, pages 204{213. IEEE, 1999.*

*[15] P. Homeier. A proof of the church-rosser theorem for the lambda calculus in higher order logic, 2001. Submitted.*

*[16] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In Proc. ICALP'01, LNCS. Springer, 2001. To appear.*

*[17] F. Honsell, M. Miculan, and I. Scagnetto. -calculus in (co)inductive type theory. Theoretical Computer Science, 253(2):239{285,* *2001.*

*[18] C. Jones. A -calculus semantics for an object-based design notation. In Proc. CONCUR'93, volume 715 of LNCS, pages 158{172. Springer, 1993.*

*[19] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. Journal of Automated Reasoning, 23(3{4):373{409,* *1999.*

*[20] T. Melham. A mechanized theory of the -calculus in HOL. Nordic Journal of Computing, 1(1):50{76,* *1995.*

*[21] D. Miller. The -calculus as a theory in linear logic: Preliminary results. In Proc. ELP'92, volume 660 of LNCS, pages 242{264. Springer, 1992.*

*[22] R. Milner. Functions as processes. Journal of Mathematical Structures in Computer Science, 17:119{141,* *1992.*

*[23] R. Milner. Communicating and Mobile Systems: the* *calculus. Cambridge*

*University Press, 1999.*

*[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Information and Computation, 100:1{77,* *1992.*

*[25] L. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.*

*[26] L. Paulson. A xedpoint approach to implementing (co)inductive de nitions.* *In Proc. CADE'94, volume 814 of LNAI, pages 148{161. Springer, 1994.*

*[27] L. Paulson, editor. Isabelle: a generic theorem prover, volume 828 of LNCS. Springer, 1994.*

*[28] C. Rockl. On the Mechanized Validation of In nite-State and Parameterized Reactive and Mobile Systems. PhD thesis, Technische Universitat Munchen, 2001.*

*[29] C. Rockl, D. Hirschko , and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the -calculus and mechanizing the theory of contexts. In Proc. FOSSACS'01, volume 2030 of LNCS, pages 364{*

*378. Springer, 2001.*

*[30] R. Vestergaard and J. Brotherston. Formalised rst-order con uence proof for the -calculus using one-sorted variable names. In Proc. RTA'01, volume 2051 of LNCS, pages 306{321. Springer, 2001.*

*[31] D. Walker. Objects in the -calculus. Information and Computation, 116(2):253{271,* *1995.*

# *A The* *Calculus*

*In nity of the set of names is expressed by an axiomatic type-class inf\_class* requiring the existence of an injection from the natural numbers to its mem- bers.

*axclass inf\_class < term*

*inf\_class "EX (f::nat=>'a). inj f"*

*In the datatype for processes, a reference to inf\_class ensures that names* always belong to it. The right-hand annotations specify a concrete syntax.

*datatype*

|  |  |  |
| --- | --- | --- |
| *'a procs = Null* |  | *(".0" 115)* |
| *| Tau* | *"('a::inf\_class) procs"* | *(".t.(\_)" [111] 110)* |
| *| Out* | *'a 'a ('a procs)* | *("\_<\_>.\_" [120, 0, 110] 110)* |

|  |  |  |
| --- | --- | --- |
| *| In* | *'a 'a ('a procs)* | *("\_[\_].\_" [120, 0, 110] 110)* |
| *| Res* | *'a ('a procs)* | *(".#\_ \_" [180, 101] 100)* |
| *| Plus* | *('a procs) ('a procs)* | *(infixl ".+" 85)* |
| *| Par* | *('a procs) ('a procs)* | *(infixl ".|" 90)* |
| *| Mt* | *'a 'a ('a procs)* | *(".[\_.=\_]\_" [100, 100, 96] 95)* |
| *| Mmt* | *'a 'a ('a procs)* | *(".[\_.~=\_]\_" [100, 100, 96] 95)* |
| *| Repl* | *('a procs)* | *(".!\_" [100] 100)* |

# *B Permutations*

*Permutation on names is completely symmetric, which carries over to pro-* cesses.

*constdefs*

*nren :: "[('a::inf\_class), 'a, 'a] => 'a" ("n{\_<->\_}\_" [0,0,199] 200) "n{x<->y}a == if a=x then y else if a=y then x else a"*

*consts*

*pren :: "[('a::inf\_class), 'a, 'a procs] => 'a procs" ("p{\_<->\_}\_" [0,0,114] 115)*

*primrec*

*"p{x<->y}.0 = .0"*

*"p{x<->y}(.t.P) = .t.p{x<->y}P"*

*"p{x<->y}(a<b>.P) = n{x<->y}a<n{x<->y}b>.p{x<->y}P"*

*"p{x<->y}(a[b].P) = n{x<->y}a[n{x<->y}b].p{x<->y}P" "p{x<->y}(.#b P) = .#n{x<->y}b p{x<->y}P"*

*"p{x<->y}(P .+ Q) = p{x<->y}P .+ p{x<->y}Q"*

*"p{x<->y}(P .| Q) = p{x<->y}P .| p{x<->y}Q"*

*"p{x<->y}(.[a.=b]P) = .[n{x<->y}a.=n{x<->y}b]p{x<->y}P"*

*"p{x<->y}(.[a.~=b]P) = .[n{x<->y}a.~=n{x<->y}b]p{x<->y}P" "p{x<->y}(.!P) = .!p{x<->y}P"*

*For sample proofs, consider the following, using induction resolved by Is-* abelle's automatic tactic auto\_tac:

*Goal "p{x<->x}P = P"; by (induct\_tac "P" 1); by (Auto\_tac);*

*qed "pren\_id";*

*Goal "{y, z} Int n P = {} --> p{y<->z}p{z<->x}P = p{y<->x}P"; by (induct\_tac "P" 1);*

*by (auto\_tac (claset() addIs [nren\_trans], simpset() delsimps [nren\_def])); qed "lemma";*

*Goal "[| y ~: n P ; z ~: n P |] ==> p{y<->z}p{z<->x}P = p{y<->x}P"; by (fast\_tac (claset() addIs [lemma RS mp]) 1);*

*qed "pren\_trans";*

# *C -Equivalence*

*The two levels of -equivalence, Al with respect to F and Alpha are de ned* in terms of an inductive set and a constant de nition, respectively.

*consts*

*Al :: "('a procs \* ('a::inf\_class) set \* 'a procs) set" Alpha :: "(('a::inf\_class) procs \* 'a procs) set"*

*syntax*

*"@Al" :: "['a procs, 'a set, 'a procs] => bool" ("\_ =a[\_] \_" [70, 0, 71] 70)*

*"@Alpha" :: "['a procs, 'a procs] => bool" ("\_ =a \_" [70, 71] 70)*

*translations*

*"P =a[F] P'" == "(P, F, P') : Al"*

*"P =a P'" == "(P, P') : Alpha"*

*inductive "Al" intrs*

*Null ".0 =a[F] .0"*

*Tau "P =a[F] P' ==> .t.P =a[F] .t.P'"*

*Out "P =a[F] P' ==> a<b>.P =a[F] a<b>.P'"*

*In "ALL x. x ~: S --> p{x<->b}P =a[insert x F] p{x<->b'}P' \*

*\ ==> a[b].P =a[F] a[b'].P'"*

*Res "ALL x. x ~: S --> p{x<->b}P =a[insert x F] p{x<->b'}P' \*

*\ ==> .#b P =a[F] .#b' P'"*

*Plus "[| P =a[F] P' ; Q =a[F] Q' |] ==> P .+ Q =a[F] P' .+ Q'"*

*Par "[| P =a[F] P' ; Q =a[F] Q' |] ==> P .| Q =a[F] P' .| Q'"*

*Mt "P =a[F] P' ==> .[a.=b]P =a[F] .[a.=b]P'"*

*Mmt "P =a[F] P' ==> .[a.~=b]P =a[F] .[a.~=b]P'"*

*Repl "P =a[F] P' ==> .!P =a[F] .!P'"*

*defs*

*Alpha\_def "Alpha == {(P, P') . EX S. finite S & P =a[F] P'}"*

*For sample proofs, consider the following, using rule induction solved by Is-* abelle's automatic tactics auto\_tac and force\_tac, or using further interac- tion:

*Goal "P =a[F] P' ==> P =a[F Un F'] P'";*

*by (etac Al.induct 1);*

*by (auto\_tac (claset() addSIs Al.intrs, simpset())); qed "Al\_un1";*

*Goal "P =a[F] P' ==> ALL Q Q' xs xs' . \*

*\ P = p{xs}Q & P' = p{xs'}Q' & \*

*\ y ~: insert z ((dom xs) Un (dom xs') Un n Q Un n Q') --> \*

*\ p{y<->z}P =a[{y, z} Un F] p{y<->z}P'"; by (etac Al.induct 1);*

*... (\* long chain of interactions \*)*

*by (REPEAT (force\_tac*

*(claset() addSDs psubst\_cases addSIs Al.intrs, simpset()) 1)); qed "lemma";*

*Goal "[| P =a[F] P' ; y ~: (n P Un n P') |] \*

*\ ==> p{y<->z}P =a[{y, z} Un F] p{y<->z}P'";*

*by (case\_tac "y=z" 1);*

*by (force\_tac (claset() addIs [Al\_insert], simpset()) 1);*

*... (\* instantiations \*) by (Force\_tac 1);*

*qed "Al\_pren\_cong1";*