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

*URL:* [*http://www.elsevier.nl/locate/entcs/volume58.html*](http://www.elsevier.nl/locate/entcs/volume58.html) *18 pages*

*A Third-Order Representation of the*

*-Calculus*

*Andreas Abel*

*Institut fur Informatik, Ludwig-Maximilians-Universitat Oettingenstr. 67, 80538 Munchen, Germany*

*Abstract*

*Higher-order logical frameworks provide a powerful technology to reason about ob- ject languages with binders. This will be demonstrated for the case of the - calculus with two di erent binders which can most elegantly be represented using a third-order constant. Since cases of third- and higher-order encodings are very rare in comparison with those of second order, a second-order representation is given as well and equivalence to the third-order representation is proven formally.*

# *1 Introduction*

*The -calculus [11,10,4], a proof theory for the implicational fragment of* classical logic, has been established as a general tool to reason about functional programming languages with control, e.g. continuations and exceptions. It is basically an extension of the -calculus by a second binder. Some of its properties like strong normalization and con uence are very fundamental for its use in functional programming and proof systems; a formal veri cation of these basic properties is therefore desirable. Human proofs are error-prone; even these which have undergone the scienti c review process. For instance, the rst published proof of con uence for the -calculus contained a aw that was only recently corrected [3].

*When reasoning about programming languages and logics with binders|* like the -calculus|the use of a higher-order logical framework can greatly reduce the size formalized proofs require. This is due to its built-in notions of

*-equivalence and substitution which make several technical lemmata obsolete*

*if a clever representation of the object language is chosen.*

*1* [*abel@informatik.uni-muenchen.de*](mailto:abel@informatik.uni-muenchen.de)

*2 This work was supported by the Graduiertenkolleg Logik in der Informatik (GKLI), Munich, and the OÆce of Technology in Education, Carnegie Mellon University.*

*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/)*.*

*To our knowledge, properties of the -calculus have not been veri ed me-* chanically so far. This article intends to provide a working point by giving two possible encodings of the -calculus in a higher-order abstract syntax. As it will be seen, the -calculus is one of the rare examples that can best be rep- resented in a way that involves a third-order constructor. This representation will enable a natural implementation of the structural or mixed substitution that has been discussed controversially in the literature [10,16,6,4].

*The remainder of this article is organized as follows: In Sect. 2 we will* introduce the -calculus in its original formulation by Parigot [11]. We will proceed to interpret it computationally in Sect. 3 by giving a small-step seman- tics and an application in exception handling. Sect. 4 contains a development of a third-order representation for the untyped and the typed -calculus to- gether with the formalization of the small-step semantics. We will point out a diÆculty with the representation and give an alternative, second-order en- coding and prove equivalence of both representations in Sect. 5. An outlook on further work will conclude this article.

# *2 Parigot's -Calculus*

*In his seminal 1992 paper Parigot [11] introduced the -calculus as a proof* theory for classical logic. In the following we will present the rst-order frag- ment which, in its computational interpretation, provides a general tool to model control in functional programming.

*The raw terms are given by this grammar:*

*M ::= x j x:M j M M j a:N (Unnamed) terms*

*N ::= [a]M Named terms*

*It is basically the untyped -calculus plus a second binder and -application* [a]M. Furthermore there is a second class of variables which we call -variables and denote by a, b and c, in contrast to the -variables denoted by x, y and z. The constructor binds a -variable in the following term; as for -variables,

*-equivalence can be de ned. In the following we will not distinguish between*

*-equivalent terms.*

*The raw terms can be typed by the following rules which we present in* natural deduction style:

*x : A*

*M : B*

*!Ix*

*M1 : A ! B M2 : A*

*!E*

*M1 M2 : B*

*a : A M : A*

*a : A*

*N :*

*raaa*

*x:M : A ! B*

*[a]M :*

*contra*

*a:N : A*

*In the rule raa the judgment N : means that the term N has no type.* The symbol denotes the absurdity, which is not considered to be a type.

*The formula A denotes the cotype of A; in the realm of logic A would be the* negated proposition A. Thus the -introduction is isomorphic to the reductio ad absurdum rule in classical logic, which states that A holds if :A leads to a contradiction.

*The contra rule is analogous to the fact in logic that :A and A entail a* contradiction. Note that there is no rule which constructs an inhabitant of a cotype. Thus, if C : A holds, C must be a -variable a. The term [a]M resulting from the application of a -variable a to a term M has no type, but it is said to be named by a and hence is called a named term.

*As closed -terms inhabit types that are isomorphic to tautologies in the* implicational fragment of intuitionistic logic, closed -terms inhabit types that are isomorphic to tautologies in the implicational fragment of classical logic. The primary example is the Pierce-type ((A ! B) ! A) ! A which is inhabited by the -term

*x(A!B)!A: aA:[a](x yA: bB:[a]y)*

*but empty in the -calculus. (For better readability we have annotated the* abstracted variables with their type.)

*2.1 Reductions*

*In the simply-typed -calculus each normal term M has the subformula prop-* erty: if x1 : A1;::: ; xn : An ` M : A then each subterm of M is an inhabitant of a type given by a subformula of A or Ai for some i. Each term M has a normal form which can be obtained by a nite number of -reductions. The reduction relation is given by the compatible closure of the -axiom.

*( x:M1)M2 ! [M2=x]M1*

*In the -calculus, we ignore negation when determining subformulas, that* is, A is considered a subformula of A. Then we can de ne a reduction relation that computes normal forms which have the subformula property as in the

*-calculus. Besides -reduction we need permutative conversions induced by* the following axiom:

*( a:N) M ! b:[ Æz:[b](z M)=a]N*

*This involves a new kind of substitution [ Æz:[b](z M)=a]N, Parigot's structural* substitution, which has the following e ect: In N, replace every subterm of the form [a]M0 by [b](M0 M) for any term M0. Our presentation of the structural substitution follows de Groote [6], who introduces internal extensions of the syntax:

*M ::= x j x:M j M M j a:N Terms*

*N ::= [C]M Named terms*

*C ::= a j Æz:N (z appears exactly once in N) Contexts*

*Structural substitution is generalized to a substitution of contexts [C=a]M.* After a context is substituted for a -variable, appearances of Æ are eliminated silently by linear -reductions:

*[ z:N]M !Æ [M=z]N:*

*Æ*

*The e ect of the -reduction is to uncover hidden -redices; it is best* explained in an example. Consider the following open term of type A:

*y : A ` ( aA!A:[a] xA:x) y : A*

*Because it contains the subterm x:x of type A ! A, this term does not* ful ll the subformula property. It is destroyed by a -redex ( x:x) y which is separated by a -abstraction. This hidden redex can be uncovered by a permutation:

*A A*

*( aA!A:[a] xA:x) y ! bA:[ ÆzA!A:[b](z y)=a]([a] xA:x)*

*= bA:[ ÆzA!A:[b](z y)] xA:x*

*!Æ b :[ x :x=z]([b](z y))*

*= bA:[b]( xA:x) y*

*! bA:[b]y*

*Note that the !Æ -reduction step counts as part of the structural substitu-* tion.

*Finally, we introduce a third kind of reduction which is analogous to -* reduction:

*[a] b:N ! [a=b]N*

*Parigot calls this reduction \renaming" of named terms. It requires a third* kind of substitution| -variables for -variables|as Ong and Steward point out [10]. However, in our formulation it is just a special case of context substitution.

*The reduction relation induced by the three axioms , and is con u-* ent. However, a straightforward adaption of the Tait and Martin-Lof parallel reduction method fails. This was overlooked in the original con uence proof [11], and only after years a correct proof was given by Baba, Hirokawa and Fujita [3]. They found that in its straightforward de nition parallel reduction does not have the diamond property. This is due to the twofold e ect of - reduction: It may create a -redex but at the same time disrupt a -redex.

*For example,*

*( a:[a] b:[a](*¸ *x:x)) y*

uu

uu

¸¸

uu

u u

uu

,*s*u

*( a:[a]( x*¸*:x)) y*

¸¸¸¸

¸¸¸

¸¸¸

¸¸¸

¸¸¸z*%*

*a:[a]( b:[a]( x:x) y) y*

¸¸¸z*$*

,*c*

*?*

*a:[a]( x:x) y*

*The -redex [a] b... is hidden after the -reduction step. Another -reduction* is required to restore it.

# *3 A Computational Interpretation*

*Since the pioneering work of GriÆn [8] it is known that control constructs* for functional programming like call/cc or Felleisen's C [7] can be typed by classical tautologies. Therefore we expect that the -calculus can be interpreted operationally such that known control constructs can be encoded as -terms. This is indeed the case|as demonstrated by Ong and Stewart

*[10] and Bierman [4]. In the following we present Ong and Stewart's call-*

*by-value -calculus, Bierman's small step semantics and an encoding of de* Groote's exception handling calculus [5] into the -calculus.

*3.1 Call-by-Value -Calculus*

*The -calculus can be extended by datatypes, case-distinction and recursion* to form the core of a programming language. This has been done by Ong and Stewart who named the resulting toy language PCF. We will stick to its core v which is the -calculus with a call-by-value reduction strategy.

*v*

*A value v is simply a -abstraction. We obtain a call-by-value strategy by* adding two rules and restricting the -axiom to values.

*( x:M) v !v [v=x]M*

*v ( a:N) ! b:[ Æz:[b](v z)=a]N*

*v*

*a:[a]M ! M (a 62 FV(M))*

*A deterministic evaluation strategy for v is given by the following small-step* semantics.

*3.2 A Small-Step Semantics*

*Bierman [4] gives a simple call-by-value small-step semantics that sheds some* light on the meaning of -abstraction and -application. Before we present

*it, we de ne redices and evaluation contexts with a single hole :*

*E ::= j EM j vE Evaluation context* R ::= vv j a:N Redex

*Lemma 3.1 (Decomposition) Every -closed unnamed term M is either a* value or can be uniquely decomposed into an evaluation context E[ ] and a redex R such that M = E[R].

*Furthermore, named terms N are always of the form [a]M. Hence, using an* environment E which maps -variables into evaluation contexts, we can spell out the small-step semantics ) by three axioms:

*(E[( x:M) v]; E ) ) (E[[v=x]M]; E )*

*(E[ a:N]; E ) ) (N; E ] fa 7! E[ ]g)*

*([a]M; E ] fa 7! E[ ]g) ) (E[M]; E ] fa 7! E[ ]g)*

*Note the invariant that the environment E binds all free -variables in M or* N.

*This semantics suggests that, in the same way as -variables are place-* holder for terms, -variables stand for contexts or continuations. Binding a

*-variable a means saving the current context in a, and applying a to a term* M means restoring the context denoted by a and continuing the evaluation of M in this context. This insight will be critical in nding the best higher-order representation later in this article.

*3.3 de Groote's Exception Handling Calculus*

*To demonstrate the capability of the -calculus to model control we present* an SML-like exception handling mechanism given by de Groote [5]. Excep- tions are added to the -calculus via two constructs: one that declares a new exception and provides a handler for it; another one that raises the exception. They are typed as follows:

*e : A*

*M : B*

*x : A*

*Me : B*

*Handle*

*e : A M : A*

*raise e(M) : B*

*Raise*

*(exception e in M handle e(x):Me) : B*

*We sketch the desired computational behavior by two examples: exception e in E[raise e(v)] handle e(x):Me ; [v=x]Me exception e in v handle e(x):Me ; v*

*Bierman gives the following translation of the two constructs into -terms.*

*Note that the encoding of raise introduces a -redex to ensure call-by-value*

*evaluation.*

*praise e(M)q = ( y: :[e]y) pMq*

*pexception e in M handle e(x):Meq = a:[a]( x:pMeq) ( e:[a]pMq)*

*The reader is invited to convince himself that this translation has the stipu-* lated evaluation behavior, e.g. by verifying that

*(pexception e in raise e(v) handle e(x):xq ; E ) ) (v; E 0):*

*Assuming the reader has gained some familiarity with the -calculus by now,* in the next section we shall proceed by discussing its formal representation in a reasoning framework.

# *4 A Third-Order Representation*

*Formal reasoning about logics and programming languages, in the following* called object theories, must take place in a framework, the meta theory. A possible choice for sucha logical framework is predicate logic: rst-order terms, made up from function symbols and variables, encode the syntactic entities of the object theory, and predicates represent properties of these entities, for example the wellformedness of a formula or the validity of a proof. However, reasoning about languages with binders in a rst-order representation is a tedious business and requires numerous technical lemmata concerning variable renaming, substitution etc. (see e.g. [15], [2]). More suitable is a logical framework with a higher-order term language which has binders itself. Then object variables can be encoded by meta variables and substitution in the object theory can be expressed by substitution in the logical framework.

*The primary candidate for a higher-order logical framework is the simply-* typed -calculus ! with -equality, in which object languages with binders can be encoded in a direct way. For example, the untyped -calculus can be represented by the following signature.

*Base types tm Terms*

*Constants lam : (tm ! tm) ! tm Abstraction app : tm ! tm ! tm Application*

*The representation function p q maps any untyped -term M into a term t of*

*the logical framework. It is de ned by recursion over M as follows:* pxq = x

*p x:Mq = lam x : tm: pMq*

*pM1 M2q = app pM1q pM2q*

*The given representation is adequate, that is, there exists a one-to-one corre-* spondence between untyped -terms and their canonical representation in the

*logical framework.*

*Theorem 4.1 (Adequacy) Let = x1 : tm;::: ; xn : tm be a context. Then*

*(i) for every untyped -term M with free variables in fx1;::: ; xng it holds* that ` pMq : tm,

*(ii) for every canonical (i.e. -normal -long) term t with ` t : tm there* exists an untyped -term M s.th. pMq = t, and

*(iii) the representation function is compositional in the sense that p[M1=x]M2q =* [pM1q =x] pM2q.

*Proof. By induction on (i.) M, (ii.) t canonical, and (iii.) M2. The cases for* abstraction x : : : refer to the induction hypothesis with an extended context ( ;x : tm). *2*

*We observe these bene ts of representing the untyped -calculus in a* higher-order logical framework like !: (i) -equivalent object terms M1 and M2 translate into equivalent terms of the logical framework and, (ii) substi- tution does not have to be implemented. More on logical frameworks can be found in [13].

*4.1 Untyped* *Calculus*

*The untyped -calculus extends the untyped -calculus by* *abstraction*

*and -application. How shall we represent the new binder? Let us analyse* the general structure of a binder bind :

*bind : ( ! ) !*

*The binder bind generates an expression of type from a context of type with a free variable of type* *We know that the binder takes a named term*

*N with a free -variable a and returns a term. Thus we add the following* constants to our signature.

*Base type nam Named terms*

*Constant mu : ( ! nam) ! tm* *abstraction*

*In this signature has to be replaced by the type of the variables that are* bound by mu. In the case of lam we had = tm since *variables just stand for*

*terms. The crucial question is: \What do -variables stand for?" The small-* step semantics suggested that they are placeholder for evaluation contexts. This is con rmed by the nature of the structural substitution which replaces contexts with contexts. We make use of the fact that contexts can be repre- sented directly in ! as functions from terms to terms and set = tm ! nam. Hence, the type of the constant mu is third-order:

*mu : ((tm ! nam) ! nam) ! tm*

*Surprisingly, we do not have to add a constant for -application; it is repre-* sented by application in !. The translation function for -terms M, N and contexts C is an extension of p q by the following de nitions:

|  |  |  |  |
| --- | --- | --- | --- |
| *p a:Nq* | *=* | *mu a : tm ! nam: pNq :* | *tm* |
| *p[C]Mq* | *=* | *pCq pMq :* | *nam* |
| *paq* | *=* | *a :* | *tm ! nam* |
| *p Æz:Nq* | *=* | *z : tm: pNq :* | *tm ! nam* |

*Theorem 4.2 (Adequacy) Let = x1 : tm;::: ; xn : tm; a1 : tm ! nam;::: ; am :* tm ! nam. Additionally to an adaption of Thm. 4.1 for it holds that

*(iv) for every named term N with free variables in it holds that ` pNq :* nam,

*(v) for every canonical term s with ` s : nam there exists a named term N* s.th. pNq = s,

*(vi) for every (not necessarily linear) context C with free variables in it* holds that ` pCq : tm ! nam,

*(vii) for every canonical term r with ` r : tm ! nam there exists a (not* necessarily linear) context C s.th. pCq = r, and,

*(viii) the representation is compositional for context substitution, that is,*

*p[C=a]Mq = [pCq =a] pMq :*

*Proof. We prove 1., 4. and 6. simultaneously by induction on M/N/C, 2.,*

*5. and 7. by induction on the canonical forms of tm, nam and tm ! nam, and*

*8. by induction on M.* *2*

*Note that we only represented general contexts adequately, not linear con-* texts C. However, all contexts we de facto write down in the encoding of reduction rules and small-step semantics will actually be of the required form. In the given encoding the structural substitution, which looks complicated in the literature (e.g. [10]), is just !-substitution for -variables (see 8.). For example,

*p[ Æz:[b](z y)=a]([a]x)q = p[b](xy)q = b (app x y) = ( z : tm :b (app z y)) x*

*= [ z : tm :b (app z y)=a](a x) = [p Æz:[b](z y)q =a] p[a]xq :*

*4.2 Typed* *calculus*

*Table 1 lists the signature 3rd which encodes the well-typed* *terms in the*

*dependently typed -calculus|the core of logical frameworks like LF [9].* To represent terms of type A we instantiate the type family tm : ty ! type to tm A. Note that in the constant declarations A and B are considered to be

*Well-typed terms:*

|  |  |  |
| --- | --- | --- |
| *ty* | *: type* | *Types* |
| *tm* | *: ty ! type* | *Typed terms* |
| *nam* | *: type* | *Named terms* |
| *i* | *: ty* | *Base type* |
| *)* | *: ty ! ty ! ty* | *Function space* |
| *lam* | *: (tm A ! tm B) ! tm(A ) B)* | *-abstraction* |
| *app* | *: tm(A ) B) ! tm A ! tm B* | *Application* |
| *mu* | *: ((tm A ! nam) ! nam) ! tm A* | *-abstraction* |
| *Reduction:* | | |
| *!* | *: tm A ! tm A ! type* | *Red. for typed terms* |
| *!n* | *: nam ! nam ! type* | *Red. for named terms* |
| *!* | *: M : tm A ! tm B: N : tm A:* |  |
|  | *app (lam M) N ! M N* | *-axiom* |
| *!* | *: M :(tm(A ) B) ! nam) ! nam: N : tm A:* |  |

*app (mu M) N ! mu a : tm B ! nam:*

*M ( z : tm(A ) B):a (app zN)) -axiom*

*!lam; !app1; !app2 : :: : lam- and app-congruences*

*(i) Direct representation (not adequate):*

*! : a : tm A ! nam: M :(tm A ! nam) ! nam:*

*a (mu M) !n M a -axiom*

*!nam : a : tm A ! nam: M ! M0*

*! aM !n aM0 nam-congruence*

*!mu : ( a : tm A ! nam: N a !n N0 a)*

*! mu N ! mu N0 mu-congruence*

*(ii) Representation with local rules (adequate):*

*!mu : ( a : tm A ! nam:*

*! : ( M :(tm A ! nam) ! nam: a (mu M) !n M a):*

*!nam : ( M; M0 : tm A: M ! M0 ! aM !n aM0 ):*

*! N a !n N0 a)*

*! mu N ! mu N0 mu-congruence*

*Table 1*

*The Signature 3rd: Well-Typed Terms and Reduction.*

*implicitly quanti ed. For example, the full type of the constant lam would be*

*A : ty: B : ty:(tm A ! tm B) ! tm(A ) B).*

*Reduction is represented by a pair of type families, ! and !n. In* part (i), the representation of the two rules ! and !nam is not adequate yet, since context C could be instantiated for a, but only -variables should be allowed. This problem can be circumvented by removing these rules and making them local : Whenever a new parameter a : tm A ! nam is introduced,

*add these rules dynamically for this speci c a. The only rule that introduces* such parameters is !mu; we replace it by the version in part (ii). Adequacy of our representation can be stated and proven in a similar way to Thm. 4.2.

*4.3 Small-Step Semantics*

*In the following we will re ne and encode the small-step semantics (M; E ) )* (M0; E 0) given in Sect. 3.2. The original formulation has a little aw: during the process of evaluation the environment E grows monotonically and accu- mulates contexts that will never be used again. We give a modi cation that uses substitution instead of environments and includes the decomposition of the subject M into evaluation context C and redex R. The -constant eval denotes the top level evaluation context.

*Values val : tm A ! type*

*v ::= x:M vlam : M : tm A ! tm B: val (lam M)*

*Evaluation Contexts*

*C ::= Æz:[eval]z eval : tm A ! nam*

*j Æz:[C](z M) z: pCq (app z pMq)*

*j Æz:[C](v z) z: pCq (app pvq z)*

*Again not each inhabitant of tm A ! nam stands for a valid evaluation con-* text. However, a judgment ecxt : (tm A ! nam) ! type which singles out the evaluation contexts can be de ned easily.

*The small-step semantics can be de ned by the four rules given below.* Formally, it maps one named term into another. We say a term M evaluates to a value v i [eval]M ) [eval]v.

*) [C](( x:M) v) ) [C]([v=x]M) ) [C]( a:N) ) [C=a]N*

*)appl*

*[ Æz:[C](z M2)]M1 ) N* [C](M1 M2) ) N

*)appr*

*[ Æz:[C](v z)]M ) N* [C](v M) ) N

*On the left hand side of ) an occurrence of [C]M is to be read as \term M in*

*context C" but on the right hand side it is just the named term [C]M. After* each step we decompose the reduct N afresh into [eval]M. This is possible since the following invariant holds: If C and M are closed and [C]M ) N, then N = [eval]M0 for a closed term M0. Obviously it holds for the rules ),

*)appl and )appr. To justify it for the rule ) we rst notice that the right*

*hand side [C=a]N is a closed named term. Hence, it must be equal to [c]M0* for some -constant c, which can only be eval.

*The representation of \)" maps a context-term pair C M into another* context-term pair C0 M0. The decomposition of a named term into context and

*term we did silently in the informal treatment is performed by the auxiliary* judgment \)n".

*) : (tm A ! nam) ! tm A ! (tm A0 ! nam) ! tm A0 ! type*

*)n : nam ! (tm A0 ! nam) ! tm A0 ! type*

*)eval : M : tm A0: (eval M) )n eval M*

*The four computation rules for \)" are given in the following. Note that*

*\)" is written in x and appears after two of its arguments.*

*) : C : tm B ! nam: M : tm A ! tm B: V : tm A: val V ! (C (M V )) )n C0 M0 ! C (app (lam M) V ) ) C0 M0*

*) : N : (tm A ! nam) ! nam: C : tm A ! nam: (NC) )n C0 M0 ! C (mu N) ) C0 M0*

*)appl : ( z:C(app z M2)) M1 ) C0 M0*

*! C (app M1 M2) ) C0 M0*

*)appr : ( z:C(app V z)) M ) C0 M0 ! val V*

*! C (app V M) ) C0 M0*

# *5 A Second-Order Representation*

*Even though we managed to show that the third-order representation of the*

*-calculus is elegant and adequate, it is not without pitfalls: As we have seen* in Sect. 4.2, the direct representation of the reduction relation is not adequate; the rules for named terms have to be locally introduced for a new parameter a : tm T ! nam. This works for ordinary reduction, but cannot be applied to a representation of parallel reduction. Consider the rule =):

*=) : a : tm T ! nam: ( b : tm T ! nam:M b =)n M0b)*

*! a(mu b:M b) =)n M0 a*

*An application of this rule has the following e ect: [a] b:M is reduced by* the -rule, and additional reductions may occur within M. The hypothesis introduces a new parameter b since we step under the binder . To make this rule local, it has to be added for each -variable that is introduced as a parameter and therefore also for b. Thus, =) has to be inserted into itself. This leads to an in nite chain of local rules and cannot be implemented.

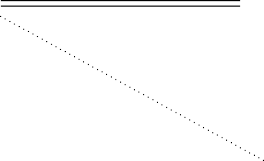
*These problems do not occur with a second-order representation, which* we present in this section. We de ne the representation function indirectly and also prove adequacy indirectly as outlined in Fig. 1. First, we de ne the

*Adequacy of the second-order representation of -terms:*

*Calc.*

*p q =*

*=*

 *=*

*p q*

z*Co3ntcan* ¸*,*

*p q =*

˛*Co1nt*

*p q =*

J *=*

z*&*J

¸*,* J

*Rep. + 3rd *

z*( 3 + 2nd)=*



˛*1+ 2nd*

*Adequacy of the second-order representation of reduction ! (Bisimulation):*

*Contcan Cont*

*L =*

J *= L0*

z*H3*

z*˛*

*0*

*H*



Ju *u*

z*H30*

*Fig. 1. Overview: Adequacy of the Second-Order Representation*

*calculus of continuations Cont, a modi cation of the -calculus which allows* general continuation terms K in a -application, now called throw K E. In principle, we make Æ an explicit constructor klam and do not treat !Æ silently any more. Furthermore, in the representation 2nd we add a base type cont for continuations. Then, we de ne the canonical expressions H *2*

*Contcan as those which only contain continuation variables, that is, no klam.* Every expression H 2 Cont can be made canonical by applying normalization

*H H0.*

*For the three representations of these calculi we can show adequacy in* a straightforward manner. Hence, in each case a one-to-one correspondence exists, and we can restrict ourselves to reason about the original calculi to obtain analogous results for their representations. Particularly, if we construct a bijection between and Contcan, we implicitly de ne an adequate second- order representation of .

*As a second part, we de ne reduction in Contcan and prove that it is* the second-order representation of the reduction in . It is suÆcient to show bisimulation of reduction in (L ! L0) and reduction in Contcan (H ! H0) as sketched in Fig. 1. The only case where we have to be careful is H ! H0 because applying structural substitution on H temporarily creates a non-canonical expression H0 on which we have to apply normalization. Thus, we incorporate normalization into *reduction.*

*In the following we will sketch the proof of adequacy of the second-order* representation of *The whole proof has been carried out in Twelf [14] and is*

*available electronically [1]. We consider it as a case study for formal reasoning* about a system by a third-order representation. To my knowledge, this is the

*rst formal proof in a representation of order strictly greater than two. This*

*may be because cases are rare where third- or higher-order representations* can be applied.

*Raw expressions:*

*E ::= x j x:E j EE j catch k:F Expressions*

*F ::= throw KE Responses*

*K ::= k j klam x:F Continuations*

*H ::= E j F j K Any Cont-term*

*Typing:*

*Signature 2nd:*

*exp : ty ! type resp : type*

*cont : ty ! type*

*lm : (exp A ! exp B) ! exp(A ) B) ap : exp(A ) B) ! exp A ! exp B catch : (cont A ! resp) ! exp A*

*throw : cont A ! exp A ! resp*

*k : A*

*F :*

*catch k:F : A*

*K : A E : A*

*throw KE :*

*x : A*

*F :*

*klam x:F : A*

*klam : (exp A ! resp) ! cont A*

*Canonical raw expressions can H:*

*can x*

*can E*

*can x:E*

*can E1 can E2*

*can E1 E2*

*can K can E can throw KE*

*can k*

*can F*

*can catch k:F*

*Normalization H H0 :*

*0 0*

*E1 E1 E2 E2*

*x y*

*E E E E*

*k l*

*0 0*

*[E=x]F F*

*0*

*throw (klam x:F ) E F*

*0*

*0*

*E E*

*1 2 1 2*

*0 0*

*0*

*F F*

*K K E E*

*0*

*0 0*

*0*

*x:E y:E*

*throw KE throw K E*

*catch k:F catch l:F*

*Translation L ==>= H from into Contcan :*

*M1 ==>= E1 M2 ==>= E2*

*x ==>= y*

*M ==>= E*

*x:M ==>= y:E*

*M1 M2 ==>= E1 E2*

*a ==>= K M ==>= E*

*[a]M ==>= throw KE*

*a ==>= k*

*N ==>= F*

*a:N ==>= catch k:F*

*Reductions H ! H0 in Contcan :*

*0*

*( x:E1 )E2 ! [E2=x]E1*

*throw k (catch l:F ) ! [k=l]F*

*[klam z:throw l (z E)=k]F F (catch k:F )E ! catch l:F*

*Table 2*

*0*

*Calculus of Continuations Cont and Canonical Fragment Contcan*

*5.1 The Calculus of Continuations Cont*

*Table 2 de nes expressions, continuations and responses (name due to Stre-* icher/Reus [16]) of Cont all of which we will refer to as raw expressions H.

*Canonical raw expressions H are those which do not contain a klam. The* judgment can H is established by recursion on H; there are congruence rules for all constructs except klam. Contcan is the quotient of Cont w.r.t. the equality induced by the axiom throw (klam x:F ) E = [E=x]F . We obtain the canonical representative H0 of a Cont-term H by applying the big-step call- by-name normalization procedure H H0.

*Lemma 5.1 (Properties of )*

*(i) If H1 H2 then can H2.*

*(ii) If H1 H2 and can H then [H=x]H1 [H=x]H2.*

*Proof (of both assertions). By induction on H1 H2.* *2*

* 1. *Bisimulation*

*The relation L ==>= H constitutes a bijective translation between terms of the*

*-calculus L and Contcan-expressions H. The following rectangle theorem* (cf. Fig. 1) states that *reductions can be simulated by Contcan-reductions.*

*Theorem 5.2 (Simulation) If L ==>= H and L ! L0 then L0 ==>= H0 and* H ! H0 for some H0.

*Proof. By induction on L ! L0. The only diÆcult case is ! for which* we need Lemma 5.3. *2*

*Lemma 5.3 (Substitution)* If

*then*

*a ==>= k*

*L ==>= H*

*and*

*x ==>= z y*

*C ==>= D0 D*

*[ Æx:C=a]L ==>= H0 [klam y:D=k]H*

*Proof. By induction on L. We spell out the hard case L = [a]M.*

*[a]M ==>= throw kE by ass.*

*M ==>= E by inversion*

*[ Æx:C=a]M ==>= E0 by ind. hyp.*

*[ Æx:C=a]([a]M) = [[ Æx:C=a]M=x]C ==>= [E0=z]D0 by assumption* E0 [klam y:D=k]E by ind. hyp.

*[E0=z]D0 [[klam y:D=k]E=y]D by assumption [E0=z]D0 throw (klam y:D) ([klam y:D=k]E) by def. of*

*= [klam y:D=k](throw k E)* *2*

*Note that the proof makes use of substitution of deductions. For example,* we show [[ Æx:C=a]M=x]C ==>= [E0=z]D0 by instantiating the deduction x ==>= z of the assumption C ==>= D0 with [ Æx:C=a]M ==>= E0.

*Analogously, it can be shown that the reduction in Contcan simulates the* reduction in the -calculus which gives us bisimulation.

# *6 Conclusion*

*We have presented the -calculus with an application and discussed two dif-* ferent encodings. The third-order representation seems to t the -calculus best since all three kinds of substitutions are reduced to substitution of the logical framework. Furthermore, we could formalize the small-step semantics economically. However, a direct representation of parallel reduction fails. The second-order representation is una ected by these problems but requires aux- iliary notions like canonical term and normalization which blow up the proofs considerably in practice.

*Two further directions of research open up from here: On the side of the* third-order representation, alternative formulations of parallel reduction have to be investigated. A concrete idea is to introduce an auxiliary judgment \C is a -variable" which makes the localizations of rules super uous.

*To improve support of the second-order representation, the logical frame-* work could be extended to allow type re nement. The type of canonical expressions would be a re nement (a subtype) of the type of expressions. The fact whether a term H is canonical could be decided and the proof of can H would be irrelevant and could be hidden. The theoretical foundations for such an extension of the logical framework have been laid by Pfenning [12].

*The third-order representation can be used to prove many properties of* the -calculus. For a start, I have formally shown soundness of the big-step semantics given by Ong and Stewart [10] wrt. to an evaluation-frames-stack small-step semantics. I expect more applications of the encoding in the future.

*Acknowledgments.*

*It was Ralph Matthes who rst interested me in the -calculus. Frank* Pfenning and Brigitte Pientka deserve my gratitude for many discussions and ideas. For comments on the draft I thank Ralph, Frank and the two anony- mous referees. Last but not least, I thank the Creator for the framework in which human life and thinking can take place.

# *References*

*[1] Abel, A., A third-order representation of the -calculus, Twelf code (2001). URL*

[*http://www.tcs.informatik.uni-muenchen.de/~abel/merlin01.tar.gz*](http://www.tcs.informatik.uni-muenchen.de/~abel/merlin01.tar.gz)

*[2] Altenkirch, T., A formalization of the strong normalization proof for System F in LEGO, in: M. Bezem and J. F. Groote, editors, Typed Lambda Calculi and Applications, TLCA'93, Lecture Notes in Computer Science 664 (1993), pp. 13{28.*

*URL*

[*http://www.tcs.informatik.uni-muenchen.de/~alti/publications.html*](http://www.tcs.informatik.uni-muenchen.de/~alti/publications.html)

*[3] Baba, K., S. Hirokawa and K. Fujita, Parallel reduction in type free - calculus, in: Proceedings of CATS 2001 (Computing: the Australasian Theory Symposium), Electronic Notes in Theoretical Computer Science 42 (2001), also appeared as Technical Report DOI-TR-177, Kyushu University, Fukuoka, Japan.*

*URL* [*http://zk.cc.kyushu-u.ac.jp/~baba/*](http://zk.cc.kyushu-u.ac.jp/~baba/)

*[4] Bierman, G. M., A computational interpretation of the -calculus, in: L. Brim,*

*J. Gruska and J. Zlatuska, editors, Proceedings of Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 1450, Brno, Czech Republic, 1998, pp. 336{345.*

*URL* [*http://www.cl.cam.ac.uk/~gmb/*](http://www.cl.cam.ac.uk/~gmb/)

*[5] de Groote, P., A simple calculus of exception handling, in: M. Dezani and*

*G. Plotkin, editors, Second International Conference on Typed Lambda Calculi and Applications, TLCA'95, Lecture Notes in Computer Science 902 (1995),*

*pp. 201{215.*

*URL* [*http://www.loria.fr/~degroote/bibliography.html#1995*](http://www.loria.fr/~degroote/bibliography.html#1995)

*[6] de Groote, P., An environment machine for the -calculus (1998), to appear in MSCS.*

*URL* [*http://www.loria.fr/~degroote/bibliography.html#wp*](http://www.loria.fr/~degroote/bibliography.html#wp)

*[7] Felleisen, M., D. P. Friedman, E. Kohlbecker and B. F. Duba, A syntactic theory of sequential control, Theoretical Computer Science 52 (1987), pp. 205{237.*

*[8] GriÆn, T. G., A formul -as-types notion of control, in: Proceedings of the Seventeenth ACM/SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL) (1990).*

*URL* [*http://www.research.att.com/~griffin/pubs.html*](http://www.research.att.com/~griffin/pubs.html)

*[9] Harper, R., F. Honsell and G. Plotkin, A Framework for De ning Logics, Journal of the Association of Computing Machinery 40 (1993), pp. 143{184.*

*[10] Ong, C.-H. L. and C. A. Stewart, A Curry-Howard foundation for functional computation with control, in: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages (1997), pp. 215{227.*

*URL*

[*http://users.comlab.ox.ac.uk/luke.ong/publications/index.html*](http://users.comlab.ox.ac.uk/luke.ong/publications/index.html)

*[11] Parigot, M., -calculus: An algorithmic interpretation of classical natural deduction, in: A. Voronkov, editor, Logic Programming and Automated Reasoning: Proc. of the International Conference LPAR'92, Springer, Berlin, Heidelberg, 1992 pp. 190{201.*

*[12] Pfenning, F., Intensionality, extensionality, and proof irrelevance in modal type theory, in: LICS 2001: IEEE Symposium on Logic in Computer Science, 2001.*

*URL* [*http://www.cs.cmu.edu/~fp/publications.html*](http://www.cs.cmu.edu/~fp/publications.html)

*[13] Pfenning, F., Logical frameworks, in: A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning (2001).*

*URL* [*http://www.cs.cmu.edu/~fp/publications.html*](http://www.cs.cmu.edu/~fp/publications.html)

*[14] Pfenning, F. and C. Schurmann, Twelf user's guide, Technical report, Carnegie Mellon University (1998).*

*URL* [*http://www.cs.cmu.edu/~twelf*](http://www.cs.cmu.edu/~twelf)

*[15] Shankar, N., A mechanical proof of the Church-Rosser theorem, Journal of the ACM 35 (1988), pp. 475{522.*

*URL* [*http://dev.acm.org/pubs/citations/journals/jacm/1988-35-3/*](http://dev.acm.org/pubs/citations/journals/jacm/1988-35-3/) *p475-shankar/*

*[16] Streicher, T. and B. Reus, Classical logic, continuation semantics and abstract machines, Journal of Functional Programming 8 (1998), pp. 543{572.*

*URL* [*http://www.mathematik.tu-darmstadt.de/~streicher/*](http://www.mathematik.tu-darmstadt.de/~streicher/)