*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) *20 pages*

*Encoding Generic Judgments:* Preliminary results

*Dale Miller* [*1*](#_bookmark0)

*Computer Science and Engineering Department, 220 Pond Lab, Pennsylvania State University Park, PA 16802-6106 USA*

*Abstract*

*Operational semantics is often presented in a rather syntactic fashion using relations* *speci ed by inference rules or equivalently by clauses in a suitable logic program- ming language. As it is well known, various syntactic details of speci cations involv- ing bound variables can be greatly simpli ed if that logic programming language has term-level abstractions ( -abstraction) and proof-level abstractions (eigenvari- ables) and the speci cation encodes object-level binders using -terms and universal quanti cation. We shall attempt to extend this speci cation setting to include the problem of specifying not only relations capturing operational semantics, such as one-step evaluation, but also properties and relations about the semantics, such as simulation. Central to our approach is the encoding of generic object-level judg- ments (universally quanti ed formulas) as suitable atomic meta-level judgments. We shall encode both the one-step transition semantics and simulation of ( nite)*

*-calculus to illustrate our approach.*

# *1 Introduction*

*The operational semantics of a programming or speci cation language is often* given in a relational style using inference rules following a small-step approach (a.k.a. structured operational semantic [[36](#_bookmark44)]) or big-step approach (a.k.a. nat- ural semantics [[13]).](#_bookmark24) In either case, algebraic ( rst-order) terms are often used to encode the language being speci ed and the rst-order theory of Horn clauses is often used to formalize and largely mechanize such semantic speci-

*cations* [*[9].*](#_bookmark21)

*For example, consider specifying a functional programming language that* has a conditional speci ed using the following inference rule (following the

*1 Email:* [*dale@cse.psu.edu*](mailto:dale@cse.psu.edu)

*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/)*.*

*natural semantics style speci cation).*

*B + true M + V (if B M N ) + V*

*B + false N + V (if B M N ) + V*

*These two inference gures can be mapped into the two rst-order Horn clauses*

*8B8M 8N 8V [B + true ^ M + V (if B M N ) + V ]*

*8B8M 8N 8V [B + false ^ N + V (if B M N ) + V ]*

*Here, the down arrow is a non-logical, predicate symbol and an expression* such as N + V is an atomic formula. A simple adequacy result shows that atomic formulas provable from such Horn clauses are exactly those for which there is a proof using the corresponding inference rules.

*If these two rules are the only rules describing the evaluation of the con-* ditional, then it should follow that if (if B M M ) + V is provable then so is M + V . In what logic can this be formalized and proved? For example, how might we prove the sequent

*(if B M M ) + V ! M + V;*

*where B, M , and V are eigenvariables (universally quanti ed)? Since such a* sequent contains no logical connectives, the standard sequent inference rules that introduce logical connective will not directly help here. One natural extension of the sequent calculus is then to add left and right introduction rules for atoms. Lars Hallnas and Peter Schroeder-Heister [7,8,40], Girard [6], and more recently, McDowell and Miller [16,14,17] have all considered just such introduction rules for non-logical constants. We outline this kind of introduction rule in the next section.

# *2 A proof theoretic form of de nitions*

*4*

*A de nition is a nite collection of de nition clauses of the form 8x [H = B],*

*where H is an atomic formula (the one being de ned), every free variable of the* formula B is also free in H, and all variables free in H are contained in the list x of variables. Since all free variables in H and B are universally quanti ed, we often leave these quanti ers implicit when displaying de nitional clauses. The atomic formula H is called the head of the clause, and the formula B is

*4*

*called the body. The symbol = is used simply to indicate a de nitional clause:*

*it is not a logical connective. The same predicate may occur in the head of* multiple clauses of a de nition: it is best to think of a de nition as a mutually recursive de nition of the predicates in the heads of the clauses.

*Given a de nition, the following two inference rules are used to introduce*

*de ned predicates. The right introduction rule is*

*! B defR ;*

*! A*

*4*

*provided that there is a clause 8x [H = B] in the given de nition such that A*

*is equal to H . The left-introduction rule is*

*fB ; ! C j 2 CSU (A; H) for some clause 8x [H = B]g*

*4*

*A; ! C defL ;*

*where the variables x are chosen to be distinct from the (eigen)variables free* in the lower sequent of the rule. The set CSU (A; H) denotes a complete set of uni ers for A and H: when the CSUs and de nition are nite, this rule will have a nite number of premises. (A set S of uni ers for t and u is complete if for every uni er of t and u there is a uni er 2 S such that is Æ for some substitution [12].) There are many important situations where CSUs are not only nite but are also singleton (containing a most general uni er) whenever terms are uni able. One such case is, of course, the rst-order case. Another case is when the application of functional variables are restricted to distinct bound variables in the sense of higher-order pattern uni cation [20,32]. In this paper, many uni cation problems will fall into this latter case.

*We must also restrict the use of implication in the bodies of de nitional* clauses, otherwise cut-elimination does not hold [39]. To that end we assume that each predicate symbol p in the language is associated with it a natural number lvl(p), the level of the predicate. We then extend the notion of level to formulas and derivations. Given a formula B, its level lvl(B) is de ned as follows:

*(i) lvl(p t ) = lvl(p)*

*(ii) lvl(?) = lvl(>) = 0*

*(iii) lvl(B ^ C) = lvl(B \_ C) = max(lvl(B); lvl(C))*

*(iv) lvl(B C) = max(lvl(B)+ 1; lvl(C))*

*(v) lvl(8x:B) = lvl(9x:B) = lvl(B).*

*We now require that for every de nitional clause 8x [p t =4 B], lvl(B) lvl(p t ).* (If the de nition is based on Horn clauses, then this restriction is trivial to satisfy since no implications would occur in the body of de nitional clauses.) Cut-elimination for this use of de nition within intuitionistic logic was proved in [14] and [17] and is modeled on proofs by Tait and Martin-Lof that use the technical notions of normalizability and reducibility. In fact, that proof also allowed the logic to contain a formulation of induction, a topic we return to later.

*We can think of de nitions as a technique to introduce logical equivalences* in such a way that we do not introduce into proof search meaningless cycles: that is, if we simply considered H B, then when proving a sequent con-

*taining H, we could replace it with B, which could then be replaced with H,* etc.

*To illustrate the strengthening of logic that can result from adding de -* nitions in this way, consider the motivating sequent above. We rst convert the two Horn clauses representing the evaluation rules for the conditional into the following de nitional clauses. (We employ the usual convention that free variables in displayed de nitional clauses are implicitly universally quanti ed around that clause.)

*(if B M N ) + V*

*4*

*= B + true ^ M + V:*

*(if B M N ) + V*

*4*

*= B + false ^ N + V:*

*This sequent then has the following simple and immediate proof.*

*B + true;M + V ! M + V initial*

*B + true ^ M + V ! M + V ^L*

*B + false;M + V ! M + V initial*

*B + false ^ M + V ! M + V ^L*

*(if B M M ) + V ! M + V defL*

*In the paper [18], the expressive strength of de nitions was studied in* greater depth. One example considered there involved attempting to capture the notion of simulation and bisimulation for labelled transition systems. In

*A*

*particular, assume that P ! P 0 is de ned via clauses to which are added*

*the two clauses in Figure 1. These two clauses are a direct encoding of the* closure conditions for simulation and bisimulation. In [18] it was proved that if the labeled transition system is nite (noetherian) then simulation and bisim- ulation coincided exactly with provability of sim P Q and bisim P Q. The restriction to noetherian transition systems is necessary since in such situ- ations, these closure clauses have unique xed points and since provability yields atoms that are true in all xed points, provability correctly charac- terizes that xed point. In transition systems with in nite paths, the least

*xed point and greatest xed point di er, so provability no longer captures*

*only the greatest xed point (simulation and bisimulation are greatest xed* points). If, however, the transition system is nitely branching, then induction can be used to characterize the greatest xed point by repeatedly applying the closure operator to the trivially true relation.

*In Section 5 we show how we can capture simulation and bisimulation for* the ( nite) -calculus in a similar style.

# *3 Should we explicitly reference provability?*

*Although we have now succeeded in giving the sequent (if B M M ) + V !* M + V a natural proof using this proof theoretic notion of de nition, it appears that we need to revisit what we really have in mind for that sequent. It seems

*4*

*sim P Q = 8*

*A8P 0: P A*

*P 0 9Q0: Q A*

*Q0 ^ sim P 0 Q0*

*4*

*!*

*bisim P Q = [8*

*A A8P 0:P !*

*A*

*A*

*P 0 9Q0:Q !*

*!*

*A*

*Q0 ^ bisim P 0 Q0] ^*

*[8A8Q0:Q ! Q0 9P 0:P ! P 0 ^ bisim Q0 P 0]*

*Fig. 1. Simulation and bisimulation as de nitions.*

*more natural that what we intend to prove is rather: \if (if B M M ) + V is* provable then M + V is provable." To explore this possibility, consider intro- ducing the predicate . that serves as an operator for provability. In this case, we now need to distinguish between two logics, one meta-level logic and one object-level logic. To do so, we shall use the type o to denote meta-level logical expressions and obj to denote object-level logical expressions. The meta-logic uses the symbols 8 of type ( ! o) ! o, 9 of type ( ! o) ! o and ^ and , both of type o ! o ! o for universal and existential quanti cation at type , for conjunction, and for implication, respectively. The object-logic

*uses the symbols V of type ( ! obj) ! obj and & and ), both of type*

*obj ! obj ! obj for universal quanti cation at type , for conjunction, and* for implication, respectively. (The type subscripts for 8 and V will often be

*dropped if they can be easily inferred or are not important.) As is the usual* convention, the expression V x will be abbreviated as simply V x.

*To encode the provability relation for the object-logic, we copy the struc-*

*ture of a logic programming interpreter following the completeness theorems* for uniform proofs and backchaining found in, say, [19], or the notion of focused proof [1,22]. Our interpreter will use the following four predicates: provability is denoted by . and has type obj ! o, backchaining is denoted by the in x symbol / and has type obj ! obj ! o, atomic of type obj ! o decides if an object-level formula is atomic, and prog, also of type obj ! o, decides if a formula is an object-level assumption (object-level logic program). A Horn clause interpreter will be written as the de nition in Figure 2. Notice that in both the . and / expression, the triangle points to the formula for which (object-level) introductions rules are considered (right-rules for . and left-rules for / ).

*The full speci cation of provability at the object level would then require* additional de nitional clauses for specifying what are atomic object-level for- mulas and what formulas constitute the object-level Horn clause speci ca- tion. Examples of such clauses are given in Figure 3. Here, + has type tm ! tm ! obj, where tm is the type of the intended programming language that we are attempting to encode the operational semantics. We can now prove the sequent

*.(if B M M ) + V ! .M + V:*

*Proposition 3.1 An atomic judgment has a proof using inference gures if* and only if it has a proof using . (Figure 2) in which the inference gures are encoded as atomic and prog clauses (as in Figure 3).

*.(G & G0) = .G ^ . G0:*

*4*

*.A = atomic A ^ prog D ^ D / A:* *A/A = atomic A:*

*4*

*4*

*(G ) D) /A = D / A ^ . G:*

*4*

*(V :Dx) /A = 9t (D t/ A):*

*4*

*Fig. 2. Interpreter for object-level speci cations.*

*atomic (M + V )*

*4*

*= >:*

*prog (V B V M V N V V [B + true & M + V (if B M N ) + V ])*

*prog (V B V M V N V V [B + false & N + V (if B M N ) + V ])*

*= >: 4*

*= >:*

*4*

*Fig. 3. Speci cation of object-level inference rules via Horn clauses.*

*Proof outline. Applications of inference gures correspond exactly to the* selection of prog clauses for backchaining over. The completeness for the treatment of the goal-reduction and backchaining steps follows from familiar completeness theorems for logic programs [27]. 2

*Having now described this interpreter, it is interesting to note that, al-* though conceptually there might be important distinctions arising from using

*., provability of M + V directly from its Horn clause speci cation or indi-* rectly via the use of this interpreter are essentially the same: (uniform) proofs in one setting map naturally to (uniform) proofs in the other setting. From a practical point of view, this distinction does not provide any proof search advantages.

*If we leave Horn clauses for a logic with universally quanti ed judgments,* then a di erence does appear. We look at this next.

# *4 -tree syntax and generic judgments*

*It is a common observation that rst-order terms are not expressive enough* to capture rich syntactic structures declaratively. In particular, such terms do not permit a direct encoding of the syntactic category of \abstraction" and the associated notions of conversion and substitution.

*4.1 Syntactic representation of abstractions*

*The encoding style called higher-order abstract syntax [35] views such abstrac-* tions as functional expressions that relying on the full power of conversion

*in a typed -calculus setting to perform substitutions. The computer systems*

*Prolog, Elf, Isabelle, and Coq, to name a few, all implement a form HOAS* and many earlier papers have appeared exploiting this style of syntactic rep- resentation [24,25,26,34]. Since the earliest papers, however, there has been a tendency to consider richer -calculi as foundations for HOAS, moving away from the simply typed -calculus setting where it was rst exploited. Trying to encode a syntactic category of abstraction by placing it within a rich function spaces can cause signi cant problems (undecidable uni cation, exotic terms, etc) that might seem rather inappropriate if one is only trying to develop a simple treatment of syntax.

*The notion of -tree syntax [28,23] was introduced to work around these* complexities. Here, -abstractions are not general functions: they can only be applied to other, internally bound variables. Substitution of general values is not part of the equality theory in the -term syntax approach: it must be coded as a separate judgment via logic. This weaker approach has a much simpler equality theory, yielding a uni cation setting (called L [20] or higher- order pattern uni cation [32,33]) which is decidable and unary. The rela- tionship between the -tree approach where abstractions are applied to only internal bound variable and HOAS where abstractions can be applied to gen- eral terms is rather similar to the distinctions made in the -calculus between

*I, which only allows \internal mobility" [38] and the full -calculus, where*

*\external mobility" is also allowed (via general substitutions). In Section 5.2,* we will see that this comparison is not accidental. In this paper, we generally view syntax as encoded using -trees.

*4.2 Generic judgments as atomic meta-level judgments*

*When using HOAS or -tree syntax representations, inference rules of the* form

*V x:Gx ;*

*A*

*are often encountered. If one were to capture this in the interpreter described* in Figure 2, there would need to be a way to interpret universally quanti ed goals. One is tempted to augment that earlier interpreter with the following clause:

*(1) .(^ x:G x) =4 8 x[.G x];*

*that is, the object-level universal quanti er would be interpreted using the* meta-level universal quanti er. While this is a common approach to dealing with object-level universal quanti cation, this encoding causes some problems when attempting to reason about logic speci cations containing generic judg- ments.

*For example, consider proving the query 8y18y2[q hy1; t1i hy2; t2i hy2; t3i];* where h ; i is used to form pairs, from the three clauses

*q X X Y:*

*q X Y X:*

*q Y X X:*

*This query succeeds only if t2 and t3 are equal. In particular, we would like* to prove the sequent

*.(^ y1 ^ y2[q hy1; t1i hy2; t2i hy2; t3i]) ! t2 = t3;*

*where t1, t2, and t3 are eigenvariables and with a de nition that consists of* the clause (1), those clauses in Figure 2, and the following clauses:

*4*

*X = X = >*

*prog (V X V Y q X X Y ) = > prog (V X V Y q X Y X) = >*

*4*

*4*

*4*

*prog (V X V Y q Y X X) = >*

*Using these de nitional clauses, this sequent reduces to*

*.(q hs1; t1i hs2; t2i hs2; t3i) ! t2 = t3;*

*for some terms s1 and s2. This latter sequent is provable only if s1 and s2* are chosen to be two non-uni able terms. This style proof is quite unnatural and it also depends on the fact that the underlying type that is quanti ed in 8y18y2 is non-empty.

*Additionally, if we use the rule (1) then whenever .(V x:G x) is provable,*

*the meta-level atomic formula .(G t) is provable for all terms t of type . While* this is likely to be appropriate when the object-language is a conventional logic, it is not likely to be appropriate when one encodes something like the

*-calculus where an object-level universal quanti er might be used to encode*

*restriction but where the presence of, say, a match pre x means that general* substitutions may not be applicable to judgments generally.

*For these reasons, the conversion of an object-level universal quanti er into* a meta-level universal quanti er in (1) must be judged inappropriate. We now look for a di erent approach.

*Consider the rule for proving a universal formula:*

*;c : ` Pc*

*` V x:P x*

*where c is an eigenvariable with the usual restriction that c is not free in the* lower sequent. Here, P is a variable of higher type, and the context, , denotes

*a set of distinct typed eigenvariables. If we see the judgment ` V x:P x*

*as \atomic" and that this is the only way to prove a universally quanti ed* formula, then this rule can be inverted: that is, if V x:P x is provable assuming

*the variables in are generic then Pc is provable assuming that c is also*

*generic. Whether or not that generic c can be instantiated and yield another* valid judgment is dependent on the object-level itself. In other words, we will require that this universally quanti ed variable acts as a bound variable but will not assume that it can be arbitrarily instantiated: using an analogy from before, we will assume that we can apply this abstraction to another abstracted variable but not to an arbitrary value.

*Thus we need to encode the object-level judgment x1;::: ; xn ` (P x1 ::: xn),* where the variables on the left are all distinct and understood as bound entirely within this judgment, as an atomic formula in our meta-logic. We mention two ways to achieve this encoding. The rst introduces a \local" binders using a family of constants, say, loc of type ( ! obj) ! obj. The above expression would be something of the form

*loc1 x1 ::: locn xn: P x1 ::: xn:*

*While this encoding is natural, it hides the top-level structure of P x1 ::: xn* under a pre x of varying length. Uni cation and matching, which are central to the functioning of the de nition introduction rules, would not be able to directly access that top-level structure. The second alternative employs a coding technique used by McDowell [14,15]. Here, one abstraction, say for a variable l of type evs (eigenvariables), is always written over the judgment and is used to denote the list of distinct variables x1;::: ; xn. Individual variables are then accessed via the projections of type evs ! and ^ of type evs ! evs. For example, the judgment x : a; y : b; z : c ` P xyz could be encoded as either the expression loca xlocb ylocc z:P xyz; or as

*l(P ( al)( b( ^l))( c( ^( ^l)))):*

*In this second, preferred encoding, the abstraction l denotes a list of variables,* the rst variable being of type a, the second being of type b, and the third of type c.

*4.3 A interpreter for generic judgments*

*An interpreter for generic judgments is displayed in Figure 4. This interpreter* generalizes the previous interpreter by allowing for the additional abstraction over evs. Here, the three meta-level predicates atomic , prog , and . all have the type (evs ! obj) ! o while / has the type (evs ! obj) ! (evs ! obj) !

*o. Notice that the technique of replacing the abstraction l: V w:(G l w))*

*4*

*. ((G l)& (G0 l . (Gl) ^ . (G0l):*

*l )) = l l*

*.l(V w:(G l w))*

*.l(Al)*

*= .l(G( ^l)(l)):*

*= atomic A ^ progD ^ (Dl) /l (Al):*

*4*

*4*

*(Al) /l (Al) = atomic A:*

*4*

*((G l) ) (D l)) /l (Al)*

*4*

*= (Dl) /l (Al) ^ .l(Gl):*

*(V w:(D l)) /l (Al)*

*4*

*= 9evs! t:(D l (t l) /l (Al)):*

*Fig. 4. An interpreter for simple generic judgments. with l:G( ^l)( l)) is really the same as replacing the judgment*

*x1;::: ; xn ` 8y(P yx1 ::: xn) with x1;::: ; xn; xn+1 ` (P x1 ::: xnxn+1):*

*Notice that this interpreter is not in L for two reasons. First, the def-* initional clause for interpreting ( l: V w:(G l w)) contains the expression

*( l:G( ^l)(l)) and the subterms ( ^l) and (l) are not distinct bound vari-*

*ables. They are, however, distinct object-level variables so it should be a rather* simple matter to extend the technical de nition of L to also allow for this style encoding of object-level variables. A second reason that this interpreter

*is not in L is the de nitional clause for backchaining over ( l: V w:(D l w))*

*since this clause contains the expression ( l:D l (t l)), which requires apply-* ing an abstraction to a general (external) term t. Such a speci cation can be made into an L speci cation by encoding object-level substitution as an ex- plicit judgment [21]. The fact that this speci cation is not in L simply means that when we apply the left-introduction rule for de nitions, uni cation may not produce a most general uni er.

*Proposition 4.1 Let n 1 and let x0: 0;::: ; xn: n be distinct variables such that (P x0 ::: xn) is an atomic formula in which the variables x0;::: ; xn are not free in P . The judgment x0;::: ; xn ` (P x1 ::: xn) has a proof using inference gures (admitting universally quanti ed premises) if and only if*

*. P (0 l*

*l) (n*

*( ^nl))*

*has a proof using the interpreter in Figure 4 in which the inference gures are* encoded as atomic and prog clauses (as in Figure 3).

*Proof outline. Applications of inference gures correspond exactly to the* selection of prog clauses for backchaining over. Here, the use of and ^ ensures that object-level eigenvariables are represented by new terms at the meta-level. For this encoding to work properly, we also assume that no constants at the object-level have types involving evs. 2

*Other judgments besides generic judgments can be encoded similarly. For* example, in [14,15], hypothetical as well as linear logic judgments were en-

*coded along these lines. The main objective in those papers is to encode an* object-level sequent as atomic judgments in a meta-logic. We focus on generic judgments here because of their relationship to abstractions within syntax.

# *5 The -calculus*

*To illustrate the use of this style representation of universal judgments, we* turn, as many others have done [28,11,3,37], to consider encoding the - calculus. In particular, we follow the presentation in [28] for the syntax and one-step operational semantics.

*5.1 Syntax*

*We shall follow the presentation of the -calculus given in [29]. We need three primitive syntactic categories: name for channels, proc for processes, and action for actions. The output pre x is the constructor out of type name ! name ! proc ! proc and the input pre x is the constructor in of type name ! (name ! proc) ! proc: the -calculus expressions x y:P and x(y):P are represented as (out x y P ) and (in x y:P ), respectively. We use j and*

*+, both of type proc ! proc ! proc and written as in x, to denote parallel*

*composition and summation, and of type (name ! proc) ! proc to denote* restriction. The -calculus expression (x)P will be encoded as n:P , which itself is abbreviated as simply x:P . The match operator, [ = ] is of type name ! name ! proc ! proc. When is written as a pre x, it has type proc ! proc. When is written as an action, it has type action. The symbols # and ", both of type name ! name ! action, denote the input and output actions, respectively, on a named channel with a named value.

*We shall deal with only nite -calculus expression, that is, expressions* without ! or de ned constants. Extending this work to in nite process ex- pressions can be done using induction, as outlined in [18] or by adding an explicit co-induction proof rule dual to the induction rule. Fortunately, the

*nite expressions are rich enough to illustrate the issues regarding syntax and*

*abstractions that are the focus of this paper.*

*Proposition 5.1 Let P be a nite -calculus expression using the syntax of* [29]. If the free names of P are admitted as constants in the meta-logic of type name then P corresponds uniquely to a -equivalence class of terms of type proc.

*5.2 One-step transitions*

*The transition semantics uses two predicates: ! of type proc ! action !*

*proc ! obj; and \* of type proc ! (name ! action) ! (name !*

*proc) ! o. The rst of these predicates encodes transitions involving free* values and the second encodes transitions involving bound values. Figure 5

*speci es the one step transition system for the \core" -calculus. Figure 6*

*provides the increment to the core rules to get the late transition system, and* Figure 7 gives the increment to the core to get the early transition system. Note that all the rules in the core system belong to the L subset of logic speci cations: that is, abstractions are applied to only abstracted variables (either bound by a -abstraction or bound by a universally quanti er in the premise of the rule). Furthermore, note that each of the increments for the late and early systems involve at least one clause that is not in L. The core system of rules has also been singled out and named I [38] since it only allows for \internal" mobility of names, that is, local (restricted) names only being passed to abstractions.

*One advantage of this style of speci cation over the traditional one [29]* is the absence of complicated side-conditions on variables: they are handled directly by the logical mechanisms described above.

*In order for this theory of one-step transitions to be interpreted by the* prover given in Figure 4, we need to take the following steps.

*Convert the inference rules of either the core, late, or early system into prog* clauses. This is straightforward (as illustrated in Section 1).

*Axiomatize the atomic predicate, which would simply be the two de nitional* clauses

*A 4*

*atomic (P ! Q) = >*

*A*

*4*

*atomic (P \* Q) = >*

*One clause for the . and one clause for / are parametrized by a type .* Here, the clause for . needs just one instance for equal to name while

*/ needs 7 di erent instances on each for set equal to name, action, proc, name ! name, name ! action, name ! proc, and name ! name ! proc.*

*Proposition 5.2 Let P*

*A*

*! Q be provable in late (resp., early) transition*

*A*

*system of [29]. If A is either or # xy or " xy then .l P ! Q is provable*

*from the clauses for the interpreter plus the clauses encoding late (resp., early)* transitions. (Here, P , A, and Q are all translated to the corresponding meta-

*# x " x*

*level expression.) If A is x(y) then .l P \* R and if A is x (y) then .l P \**

*R. Here, R is the meta-level representation of the -abstr*[*action*](#_bookmark7) *of* [*y*](#_bookmark9) *over Q.*

*Proof Outline. Follows almost directly from Propositions 4.1 and 5.1. The* induction needs to be strengthen slightly to handle the case where the bound variable l in .l are free in the judgment, which can happen, of course, when a universally quanti ed goal is interpreted. 2

*A*

*Since the types of P and P 0 are di erent in the expression P \* P 0, we*

*cannot immediately form the transitive closure of this relationship to give a* notion of a sequence of transitions. It is necessary to lower the type of P 0 rst by applying it to a term of type name. How this is done, depends on what we

*A*

*P ! Q*

*A*

*match*

*A*

*P \* Q*

*A*

*match*

*:P ! P*

*[x = x]P ! Q*

*[x = x]P*

*\* Q*

*A*

*P ! R*

*A*

*sum*

*A*

*Q ! R*

*A*

*sum*

*A*

*P \* R*

*A*

*sum*

*P + Q ! R*

*A*

*Q \* R*

*sum*

*A*

*P + Q \* R*

*A*

*P + Q ! R*

*A*

*P ! P 0*

*par*

*A*

*P jQ ! P 0jQ*

*P + Q \* R*

*A*

*Q ! Q0*

*A*

*P jQ ! P jQ0*

*A*

*par*

*P \* M*

*A*

*P jQ \* n(M njQ)*

*A*

*n(Pn ! P 0n)*

*V*

*res*

*A*

*par*

*Q \* N*

*A*

*P jQ \* n(P jN n)*

*A*

*V n(Pn \* P 0n)*

*A*

*par*

*res*

*n:P n ! n:P 0n n:P n \* m n:(P 0mn)*

*"xy* *out x y P ! P*

*output*

*"xy*

*#x*

*in x M \* M*

*input*

*V y(My ! M 0y)*

*"x*

*y:M y \* M 0*

*open*

*#x "x "x #x*

*P \* M Q \* N*

*P jQ ! n:(M njN n)*

*close*

*P \* M Q \* N*

*P jQ ! n:(M njN n)*

*close*

*Fig. 5. The core -calculus in -tree syntax.*

*#x "xy "xy #x*

*P \* M Q ! Q0*

*P jQ ! (My)jQ0*

*L-com*

*P ! P 0 Q \* N*

*P jQ ! P 0j(Ny)*

*L-com*

*Fig. 6. The additional rules for late -calculus.*

*"xy*

*#xy*

*#xy*

*in x M ! My*

*E-input*

*#xy*

*"xy*

*P ! P 0 Q ! Q0*

*E-com*

*Q ! Q0 P*

*! P 0*

*E-com*

*P jQ ! P 0jQ0 P jQ ! P 0jQ0*

*Fig. 7. The additional rules for early -calculus.*

*sim*

*4*

*(P l) (Ql*

*Al*

*A8P 0[. (Pl*

*Al*

*P 0l) 9Q0 . (Ql*

*Q0l)^*

*l ) = 8*

*l !*

*l !*

*#(Xl)*

*siml (P 0l) (Q0l)] ^ #(Xl)*

*8X8P 0[.l(Pl \* P 0l) 9Q0 .l(Ql \* Q0l)^*

*8w siml (P 0lw) (Q0lw)] ^*

*"(Xl) "(Xl)*

*8X8P 0[.l(Pl \* P 0l) 9Q0 .l(Ql \* Q0l)^*

*siml (P 0( ^l)( l)) (Q0( ^l)( l))]*

*Fig. 8. De nitional clause for simulation of -calculus*

*are trying to model. In the next section, we consider modeling simulation.*

*5.3 Simulation of -expressions*

*For simplicity, we shall consider only simulation and not bisimulation: extend-* ing to bisimulation is not diÆcult (see Figure 1) but does introduce several more cases and make our examples more diÆcult to read.

*Figure 8 presents a de nitional clause for simulation. Here, the meta-* logical predicate sim is of type (evs ! proc) ! (evs ! proc) ! o and again, we abbreviate the expression sim( l:P l)( l:Ql) as siml(P l)(Ql). Here, X has type evs ! name, P has type evs ! proc, and P 0 has two di erent types, evs ! proc and evs ! name ! proc. Since the only occurrence of is such that is name, we shall drop the subscript on . Notice also that for this set of clauses to be successfully strati ed, the level of sim must be strictly greater than the level of all the other predicates of the interpreter (which can all be equal).

*Notice also that sim is a meta-level predicate while ! and \* are* object-level predicates. This is a required separation since simulation needs to encompass the provability of these one-step translation relations. This is di erent from the encoding in [18] (Figure 1) since no universal judgments were needed to encoded CCS and hence the object/meta-level distinction was not needed.

*The rst conjunct in the body of the clause in Figure 8 deals with the case* where a process makes either a step or a free input or output action. In these cases, the variable A would be bound to either l: (in the rst case) or

*l: # (N l)(M l) or l: " (N l)(M l), in which cases, N and M would be of*

*the form l: ( ^il) for some non-negative integer i.*

*The last two cases correspond to when a bounded input or output action* is done. In the case of the bounded input (the second conjunct), a universal quanti er of the meta-logic, 8w, is used to instantiate the abstractions (P 0 and Q0), whereas in the bounded output case (the third conjunct), a universal

*quanti er of the object-level is emulated: such an internal quanti er is im-* mediately replaced by using a new variable in the context, via the use of and ^. This one de nition clause thus illustrates important distinctions about meta-level and object-level: in particular, the observation that simulation is encoded as a meta-level predicate and not an object-level predicate (as with the one-step predicates) and that the universal quanti ers in both logics each have their applications and should not be confused.

*5.4 Modal Logics for -calculus*

*To further illustrate the ease of handling and encoding binding structures, we* now encode the modal logic for the for -calculus given in [30] (of necessity, we consider only binary conjunctions instead of general, indexed conjunc- tions). We rst introduce the new type assert to denote assertion forms and then introduce the following constructors of this type: true : assert for true,

*and : assert ! assert ! assert for conjunction, not : assert ! assert for*

*negation, h = i : name ! name ! assert ! assert for the match modal,* h i : action ! assert ! assert for the possibility modal for non-binding ac- tions, and the following four modal operators used to encode the possibility of a bound actions: h# i ; h# iL ; h# iE ; h" i : name ! (name ! assert) ! assert. The rst three of these modals are used to code the \basic", \late", and \early" versions of the bounded input pre x while the forth encodes the bounded output action. Natural numbers are encoded as the type nat with constants z : nat and succ : nat ! nat. The satisfaction relation is de ned using two predicates: j= at type (evs ! proc) ! (evs ! assert) ! o as well as at the type nat ! (evs ! proc) ! (evs ! assert) ! o. This extra argu- ment is used to help stratify this de nition in the presence of negation in the assertion language. The predicate depth is of type (evs ! assert) ! nat ! o

*and the expression depth( l:Bl) N is abbreviated as simply depthl(Bl; N ).* This predicate holds if N is an upper bound on the nesting of negations in

*the (Bl) formula: this number is used to pick a suitable level to start the* use of the strati ed version of satisfaction. As we have done before, the ex- pression j= ( l:P l) ( l:Bl) is abbreviated as Pl j=l Bl while the expression j=i ( l:P l) ( l:Bl) is abbreviated as Pl j=i Bl.

*l*

*To properly stratify the de nition in Figure 9, a slight generalization to* the de nition of levels for predicates needs to be made. In particular, we need to see the expression j=i as a predicate of level i and then give the predicate j= the level !. In other words, levels need to be generalized beyond nite ordinal.

# *6 Related and future work*

*Of course, the value of this approach to encoding the -calculus comes, in* part, from the ability to automate proofs using such de nitions. For example,

*Pl j=*

*Bl = 9i:depth (Bl; i) ^ Pl j=i Bl*

*l l l*

*4*

*4*

*depthl(true;N ) = >*

*depthl((B1l) and (B2l);N ) = depthl(B1l; N ) ^ depthl(B2l; N )*

*4*

*4*

*depthl(not(Bl); succ N ) = depthl(Bl; N )*

*depthl(hXl = Y liBl; N ) = depthl(Bl; N )*

*4*

*4*

*depthl(hAliBl; N ) = depthl(Bl; N )*

*4*

*depthl(h# AliBl; N ) = depthl(B( ^l)( l);N )*

*depthl(h# AliEBl; N ) = depthl(h# AliLBl; N ) =*

*4*

*4*

*depthl(B( ^l)( l);N )*

*depthl(B( ^l)( l);N )*

*depthl(h" AliBl; N ) = depthl(B( ^l)( l);N )*

*4*

*Pl j=i true 4*

*l = >*

*Pl j=i (B l) and (B l) =4 Pl j=i B l ^ P j=i B l*

*l 1 2 l 1 l 2*

*Pl j=i+1 not(Bl Pl j=i Bl) ?*

*4*

*l ) = ( l*

*Pl j=i hXl = XliBl =4 P j=i B*

*l*

*l*

*Pl j=i hAliBl 4*

*Al*

*. Pl*

*P 0l ^ P 0l j=i Bl*

*l = l ! l*

*Pl j=i h# AliBl 4*

*Al*

*P 09z(. Pl*

*0 ^ P 0lz j=i Blz)*

*l = 9 l \* P l l*

*Pl j=i h# AliEBl =4*

*Al*

*8z9P 0(. Pl*

*0 ^ P 0lz j=i Blz)*

*l l \* P l l*

*Pl j=i h# AliLBl =4*

*Al*

*9P 08z(. Pl*

*0 ^ P 0lz j=i Blz)*

*l l \* P l l*

*Pl j=i h" AliBl 4*

*Al*

*P 0(. Pl*

*0 ^ P 0( ^l)( l) j=i B( ^l)( l))*

*l = 9 l \* P l l*

*Fig. 9. De nition of a process satisfying an assert formula. one would hope that the sequent*

*assertl(Bl); Pl j=l Bl; siml(P l)(Ql) ! Ql j=l Bl;*

*would have a simple, natural proof (where the predicate assertl( ) describes* a subset of the modal logic that would correspond to simulation). Jeremie Wajs and the author are working on a tactic-style theorem prover for a logic with induction and de nitions. This system, called Iris, is written entirely in Nadathur's Teyjus implementation [31] of Prolog and appears to be the

*rst theorem proving system to be written entirely using higher-order abstract*

*syntax (parser, printer, top-level, tactics, tacticals, etc). Example proofs that*

*we have done by hand come out as expected: they are rather natural and* immediate, although the encoding of eigenvariable context as a single abstrac- tion makes expressions rather awkward to read. Fortunately, simple printing and parsing conventions can improve readability greatly. Given that the in- terpreter in Figure 4 is based on Horn clauses de nitions, it is possible to employ well known induction principles for Horn clauses to help prove such properties.

*There are various other calculi, such as the join and ambient calculi, in* which names and name restriction are prominent and we plan to test this style of encoding with them. Generic judgments have been used to model names for references and exceptions [22,2] so it would be interesting to see if this style of encoding can adequately help in reasoning about programming language semantics containing such features. Comparing this particular encoding of the -calculus with those of others, for example, [11,3,37], should be done in some detail.

*Finally, the approach to encoding syntax and operational semantics used* here is strongly motivated by proof theoretic considerations. There has been much work lately on using a more model-theoretic or categorical-theoretic approaches for such syntactic representations, see for example [4,5,10]. Com- paring those two approaches should be quite illuminating.

*Acknowledgements.*

*Catuscia Palamidessi has made a number of comments on a draft of this* paper. Anonymous reviewers also made several corrections to the presentation of this paper. This work is funded in part by NSF grants CCR-9912387, CCR- 9803971, INT-9815645, and INT-9815731.

# *References*

*[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. Journal of Logic and Computation, 2(3):297{347, 1992.*

*[2] Jawahar Chirimar. Proof Theoretic Approach to Speci cation Languages. PhD thesis, University of Pennsylvania, February 1995.*

*[3] Jolle Despeyroux. A higher-order speci cation of the -calculus. In Proc. of the IFIP International Conference on Theoretical Computer Science, IFIP TCS'2000, Sendai, Japan, August 17-19, 2000., August 2000.*

*[4] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In 14th Annual Symposium on Logic in Computer Science, pages 193{202. IEEE Computer Society Press, 1999.*

*[5] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In 14th Annual Symposium on Logic in Computer Science, pages 214{*

*224. IEEE Computer Society Press, 1999.*

*[6] Jean-Yves Girard. A xpoint theorem in linear logic. Email to the* [*linear@cs.stanford.edu*](mailto:linear@cs.stanford.edu) *mailing list, February 1992.*

*[7] Lars Hallnas and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. i. Clauses as rules. Journal of Logic and Computation, pages* *261{283, December 1990.*

*[8] Lars Hallnas and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. ii. Programs as de nitions. Journal of Logic and Computation,* *1(5):635{660, October 1991.*

*[9] John Hannan. Extended natural semantics. J. of Functional Programming,* *3(2):123{152, April 1993.*

*[10] M. Hofmann. Semantical analysis of higher-order abstract syntax. In 14th Annual Symposium on Logic in Computer Science, pages 204{213. IEEE* *Computer Society Press, 1999.*

*[11] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)inductive type theory. Theoretical Computer Science, 253(2):239{285, 2001.*

*[12] G erard Huet. A uni cation algorithm for typed -calculus. Theoretical* *Computer Science, 1:27{57, 1975.*

*[13] Gilles Kahn. Natural semantics. In Proceedings of the Symposium on* *Theoretical Aspects of Computer Science, volume 247 of LNCS, pages 22{39. Springer-Verlag, March 1987.*

*[14] Raymond McDowell. Reasoning in a Logic with De nitions and Induction. PhD* *thesis, University of Pennsylvania, December 1997.*

*[15] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract* *syntax in a logical framework. To appear in the ACM Transactions on Computational Logic.*

*[16] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science, pages 434{445, Warsaw, Poland, July 1997. IEEE Computer Society Press.*

*[17] Raymond McDowell and Dale Miller. Cut-elimination for a logic with de nitions and induction. Theoretical Computer Science, 232:91{119, 2000.*

*[18] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. Theoretical Computer Science, 197(1-2), 2001. To appear.*

*[19] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, Logic and Computer Science, pages 329{359. Academic Press, 1990.*

*[20] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple uni cation. Journal of Logic and Computation, 1(4):497{ 536, 1991.*

*[21] Dale Miller. Uni cation of simply typed lambda-terms as logic programming. In Eighth International Logic Programming Conference, pages 255{269, Paris,* *France, June 1991. MIT Press.*

*[22] Dale Miller. Forum: A multiple-conclusion speci cation language. Theoretical Computer Science, 165(1):201{232, September 1996.*

*[23] Dale Miller. Abstract sybtax for variable binders: An overview. In John Lloyd and et. al., editors, Computational Logic - CL 2000, Springer LNAI 1861, pp. 239{253 (2000).*

*[24] Dale Miller and Gopalan Nadathur. A computational logic approach to syntax and semantics. Presented at the Tenth Symposium of the Mathematical Foundations of Computer Science, IBM Japan, May 1985.*

*[25] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, pages 247{255. Association for* *Computational Linguistics, Morristown, New Jersey, 1986.*

*[26] Dale Miller and Gopalan Nadathur. A logic programming approach to* *manipulating formulas and programs. In Seif Haridi, editor, IEEE Symposium on Logic Programming, pages 379{388, San Francisco, September 1987.*

*[27] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic, 51:125{157, 1991.*

*[28] Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. In Pierpaolo Degano, Roberto Gorrieri, Alberto Marchetti-Spaccamela, and Peter* *Wegner, editors, ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective, volume 31. ACM, Sep 1999.*

*[29] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. Information and Computation, pages 41{77, 1992.*

*[30] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. Theoretical Computer Science, 114(1):149{171, 1993.*

*[31] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus|a compiler and abstract machine based implementation of Lambda Prolog. In*

*H. Ganzinger, editor, Proceedings of the 16th International Conference on Automated Deduction, pages 287{291, Trento, Italy, July 1999. Springer-Verlag LNCS.*

*[32] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, LICS91, pages 342{349. IEEE, July 1991.*

*[33] Tobias Nipkow. Functional uni cation of higher-order patterns. In M. Vardi, editor, LICS93, pages 64{74. IEEE, June 1993.*

*[34] Lawrence C. Paulson. The foundation of a generic theorem prover. Journal of Automated Reasoning, 5:363{397, September 1989.*

*[35] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In Proceedings* *of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, pages 199{208. ACM Press, June 1988.*

*[36] G. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.*

*[37] C. Rockl, D. Hirschko , and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In Proceedings of FOSSACS'01, LNCS, 2001.*

*[38] Davide Sangiorgi. -calculus, internal mobility and agent-passing calculi. Theoretical Computer Science, 167(2):235{274, 1996.*

*[39] Peter Schroeder-Heister. Cut-elimination in logics with de nitional re ection. In D. Pearce and H. Wansing, editors, Nonclassical Logics and Information Processing, volume 619 of Lecture Notes in Computer Science, pages 146{171. Springer-Verlag, 1992.*

*[40] Peter Schroeder-Heister. Rules of de nitional re ection. In M. Vardi, editor, Eighth Annual Symposium on Logic in Computer Science, pages 222{232. IEEE Computer Society Press, IEEE, June 1993.*