Natural semantics on the computer

Dominique Clement, Joelle Despeyroux, Thierry Despeyroux, Laurent Hascoet, Gilles Kahn

▶ To cite this version:

Dominique Clement, Joelle Despeyroux, Thierry Despeyroux, Laurent Hascoet, Gilles Kahn. Natural semantics on the computer. [Research Report] RR-0416, INRIA. 1985. inria-00076140

HAL Id: inria-00076140
https://inria.hal.science/inria-00076140

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports de Recherche

N° 416

NATURAL SEMANTICS ON THE COMPUTER

Dominique: CLÉMENT
Joelle. DESPEYROUX
Thierry. DESPEYROUX
Laurent. HASCOET
Gilles. KAHN

Juin 1985

Natural Semantics on the Computer

D.Clément, J.Despeyroux, T. Despeyroux, L. Hascoet, G. Kahn

Sophia-Antipolis, June 1985

Abstract

Defining semantics of programming languages with the help of structural axioms and inference rules has been advocated by Plotkin. We call the method Natural Semantics and show that it can be implemented on the computer. Several examples in static semantics, translation and dynamic semantics are worked out in full. They show the power and elegance of the method, as well as its intuitive appeal.

All examples discussed in this paper have been mechanically translated into running programs. Hence Natural semantics seems a strong candidate for building semantically-based programming environments.

Résumé

G. Plotkin s'est fait l'avocat d'une méthode de définition sémantique des langages de programmation à l'aide d'axiomes et de règles d'inférence structurels. Nous donnons à cette méthode le nom de Sémantique Naturelle et nous montrons qu'elle peut être implantée sur ordinateur. Nous traitons en détail plusieurs exemples concernant des spécifications de sémantique statique, de traduction et de sémantique dynamique. Ces exemples mettent en valeur la puissance, l'élégance et l'aspect intuitif de cette approche.

Tous les exemples traités dans cet article ont été compilés et sont exécutables. Par suite, il semble que cette méthode de Sémantique Naturelle puisse servir de fondement pour un système dérivant intégralement un environnement de programmation d'une simple description sémantique.

PAPIER RECUPERE ET RECYCLE

Natural Semantics on the Computer

D.Clément, J.Despeyroux, T. Despeyroux, L. Hascoet, G. Kahn

INRIA Sophia-Antipolis
SEMA

Abstract

Defining semantics of programming languages with the help of structural axioms and inference rules has been advocated by Plotkin. We call the method Natural Semantics and show that it can be implemented on the computer. Several examples in static semantics, translation and dynamic semantics are worked out in full. They show the power and elegance of the method, as well as its intuitive appeal.

All examples discussed in this paper have been mechanically translated into running programs. Hence Natural semantics seems a strong candidate for building semantically-based programming environments.

INTRODUCTION

The formal description of programming languages has developed along two general lines: attribute grammars –to specify static semantics and translations– and denotational semantics –mostly to specify dynamic semantics. Both formalisms, while extremely useful, have their deficiencies.
The major deficiencies of attribute grammars are:

- Specifications based on attribute grammars often result in heavy – seemingly very low level – notations.
- Attributes are attached to single tree nodes rather than tree patterns; as a consequence structural information obscures attributes.
- The formalism seems more appropriate for static calculations than for dynamic execution.
- Semantic analysis of attribute grammars is difficult due to the low level of the formalism.

The major deficiencies of denotational semantics are:

- For static semantics, denotational semantics equations are clumsy and the ways to specify tree traversal are not very elegant.
- It seems difficult to describe parallel constructs in the dynamic semantics of a language.
- Pure denotational semantic definitions may result in overspecification.

This research has been partially supported under ESPRIT, Project 348

As a result, an approach advocated by Gordon Plotkin in his Lecture Notes at Aarhus University [Plotkin], *A Structural Approach to Operational Semantics*, has been studied and experimented with extensively. This approach consists in presenting an axiomatization –via axioms and inference rules guided by the structure of the language– of various predicates such as, for example:

- In this context, this program fragment has this type.
- In this context, this program fragment can be translated into that program fragment.
- In this context, this program fragment allows transition from this state to that state.

The method, for historical reasons, is called Structural Operational Semantics. We prefer to call it Natural Semantics because of its intuitive appeal and its Natural Deduction flavor. It retains the best aspects of earlier methods:

- Semantics is defined recursively on the structure of the formalism (as is the case in denotational semantics).
- The definition is declarative (BNF, Attribute Grammars, Predicate logic).
- Pattern Matching, Unification, Overloading are used intensively.

Furthermore, the method exhibits progress on several key points:

- The definitions are short, readable, elegant. With some typesetting effort, they look very intuitive.
- Several concepts from attribute grammars can be recovered (such as incremental computation).
- Interfacing a definition of this kind with recursive semantic equations or abstract algebraic specifications seems feasible.
- Static semantics and translation can easily be expressed.
- Specifying concurrent behaviour is reasonably easy and natural.

1. A computer formalism

To start with, we must design a computer formalism that approximates the style of structural operational semantics. This work results in a language that, for the moment, we call TYPOL, because initially we used it mostly to specify type-checking. This language is a first attempt, and it is clear that several improvements are still needed.

1.1. Rules

A TYPOL program is an *unordered* collection of rules. A rule has basically two parts, a Numerator and a Denominator. Certain variables may occur both in the numerator and the denominator. These variables allow a rule to be instantiated. Of course, all occurrences of a given variable must be substituted by the same formula.

The numerator of a rule is again an *unordered* collection of predicates. Intuitively, if all of them hold, then the denominator, a single predicate, holds. More formally, from proof trees of the numerators, we can obtain a proof tree of the denominator.

A rule is thus similar to a Prolog clause. But we notice already two differences: the ordering of the rules does not matter, the ordering of the predicates on the numerator doesn't either. On the other hand, the notion of a Prolog variable is identical to the notion of a TYPOL variable, but for the fact that TYPOL variables are typed.

1.2. Sequents

Predicates are divided into two kinds: *sequents* and *conditions*. The denominator of a rule can only be a sequent. On the numerator, sequents are distinguished from conditions, and the fact that they usually occur first doesn't imply any notion of sequencing with respect to the conditions. A sequent has two parts, an *antecedent* and a *consequent*. Traditionally, we use the turnstile symbol $\vdash$ to separate these parts. The consequent may have several forms, indicated by various infix symbols. For the moment, we handle only one symbol, colon and use it to mean "has type", "translates to", "leads to". A refinement under consideration will include several distinct symbols.

A rule that contains no sequent on the numerator is called an *axiom*.

1.3. Conditions

Very often, a rule is subject to restrictions. A variable may not occur free somewhere, a value must be boolean, some relation must hold between two variables, for example. Conditions are, for the moment either predefined conditions, or equalities of the form "variable = term" or conditions that are axiomatized by another set of TYPOL rules. Typical predefined conditions are ISVAR(x), ISIN(l,x), SAMEVAR(x,y).

It is not clear yet what language or languages are adequate to define conditions.

1.4. Actions

Actions may be attached to rules, in a manner that is reminiscent of the way actions may be attached to grammar-rules in YACC. Actions are triggered only after an inference rule is considered applicable, and actions may need to get hold of the values bound to the variables of the rule. An action cannot under any circumstance interfere with the applicability of the rules. Typical use of actions concern: tracing in various ways the inference system, emitting and filtering error messages, etc...

It is clear that actions will have to be written in a variety of programming languages, at various levels of abstraction. Therefore, only the communication mechanism between TYPOL and actions has to be defined. This remains to be done.

In terms of style, actions should be used with parcimony. For example, when specifying a translation, it seems far more elegant to axiomatize it rather than have the actions perform the generation of the output. On the other hand, in the context of type-checking, it seems more appropriate to have actions filter error messages, rather than introduce strange type values to handle various erroneous situations.

1.5. Rule sets

Some structure must be introduced in a collection of rules, if only to separate different semantic concerns. For example, in a specification of types, one wishes to distinguish structural rules of consistency, management of scope and the properties of type values. To this end, rules may be grouped into sets, with a given *name*. Sets of rules may collect together rules or, recursively, rule sets. For the moment, no scope has been attached to this bracketing mechanism. When one wishes to invoke a sequent that is axiomatized in a set of rules that is not the textually enclosing one, the name of that set appears as a superscript of the turnstile symbol in the sequent.

Further bracketing is necessary, to group rules that work together. This could appear as brace, together with some optional name. This name now would be only an annotation. This situation is similar to Algol-like languages, where we may use both procedures and anonymous blocks.

1.6. Handling special cases

Rule sets are unordered. That means that, if two rules may apply, we need to find an *intrinsic* reason to prefer one rule over another. Many experiments were needed before settling on such a rule. Since our method is structural, the sequent in the denominator of a rule will always, on the right hand side of the turnstile (i.e. in the consequent) contain a syntactic term. In fact, in the case of a translation specification, there may be several such terms: then we shall select the leftmost one. Let us call this term the *subject* of the rule. If the subjects of all rules in a rule set are incompatible, (i.e. they have no common instances), then only one rule may apply at any time. If two rules have identical subjects, then the selection of what rule applies is based on their numerators.

The case of interest is when two rules have compatible (unifiable) subjects. Consider first the case when one rule, say R1, has a driving term t1 that is an instance of the subject of a rule R, say t. This indicates the rule R1 is a *special case* with respect to rule R. The intent of the specification is that rule R1 should be preferred to rule R, because it is a more precise rule. If we accept this very intuitive understanding, then we are faced again with the problem of choosing between two distinct special cases R1 and R2 of a rule R, whose subjects may be compatible. What is aimed for is the property that there always be a most specific rule to select.

To achieve this, we impose on TYPOL programs a "compile time" condition: within a rule set, the set of subjects should be *closed* under unification. In other words, if we think of the relation "$t1$ is an instance of t" as an ordering relation, ($t1 \geq t$) any two compatible subjects should have a least upper bound.

If this condition is satisfied, there always exists a most specific rule to apply. In practice, except when writing pretty-printers, we have found that this compile time constraint induces very rarely the need to add a new rule. On the other hand, the ability to handle easily special cases is a key factor in the elegance of TYPOL. It is used repeatedly in all aspects of semantic specifications.

1.7. Declarations

A set of rules may contain declarations. These declarations introduce names for the variables in the set, and give them a type. For the moment this type is just an identifier, so that variables may be grouped into sets of identical type, i.e. a *private* type as it is called usually.

Of particular importance are variables that denote subtrees of a given language L. If a variable is declared of type L, that means that it stands for a subtree belonging to the abstract syntax of L. This abstract syntax is in general imported through a *use* clause.

The declarations are used by the TYPOL pretty-printer, that may assign different fonts for variables of different types, in accordance with standard mathematical style. They are also to be used by the TYPOL type-checker.

1.8. Use clauses

A use clause serves initially to import all of the abstract syntax constructors of a given language. This allows identification, in sequents, of the *tree* patterns, and the checking of their validity. In fact, as we shall see in the definition of the language KH, an abstract syntax is a collection of overloaded constructors. So, by extension, a use clause allows to import any collection of constructors.

Since several use clauses may introduce the same constructor name, means are provided to rename of a constructor as it is being imported. However, overloading resolution in TYPOL programs should, we hope, limit the need for this facility.

Identification of languages and of constructor collections is useful for type-checking and pretty-printing. Indeed, a use clause should also import the pretty-printing rules that are used for a collection of operators, and these rules should supersede default rules for pretty-printing general terms.

1.9. Overloading

The type-checking rules for TYPOL are still under design. An essential element however, learned from experience with denotational semantics and other mathematical formalisms, is the need to use overloading. This avoids inventing many new, insignificant, names.

Overloading is found in two places in TYPOL. First, the turnstile symbol is heavily overloaded. Within a set of rules, the turnstile is a reference to the immediately (textually) enclosing set. A comparable situation would occur in an Algol-like language if one would not have to name the recursive function that one is in the process of defining. Furthermore, we wish to avoid a proliferation of names like: check-declaration, check-statement, check-expression. So we allow overloading of sequents based on the type of the arguments occurring in the sequent, whether to the left or to the right of the turnstile symbol.

As a result, we achieve a style of expression that is very compact, but still readable. An idea of this kind has, to our knowledge, already been put forward by Ravi Sethi, in the context of Denotational Semantics.

The second occurrence of overloading comes naturally with abstract syntax. Constructors, by necessity, must be overloaded, if we wish to enforce well-formedness of abstract syntax trees: an assignment statement, for example, allows expressions on the right hand side, and a subset of expressions on the left hand side, those that denote locations. Hence even the simplest constructor, **Identifier**, must be overloaded, to appear on both sides of an assignment statement, if we want to have a notion of syntactic types. Fortunately, resolving this kind of of overloading is completely trivial, as we see later in exemple KH. But if we allow internal overloading within one abstract syntax, we might as well try to allow it between two different abstract syntaxes, in the fairly rare case where we deal with several languages simultaneously, that share a constructor's name.

1.10. Pretty-printing TYPOL Programs

TYPOL programs are entered in a standard ASCII format, mostly under the MENTOR syntax oriented editor [Mentor]. However, we have built a pretty-printer that gives a far more pleasant and familiar outlook to these programs. The pretty-printer generates TeX input, which is then processed by TeX and perused on a high resolution display. In this way, we can obtain properly computed blank spaces, nice horizontal bars and a choice of fonts for very little work of the user.

Variable declarations *are used* by the pretty-printer. A font is attached as an annotation to a given declaration. This font is used for all the variables bound by this declaration. Second, after overloading resolution if necessary, we use separately designed pretty-printers attached to the modules imported via use clauses. If a pretty-printer has not yet been designed for a given formalism, then we fall back on a default pretty-printer for abstract syntax trees. This organization is very modular, and we reap nice benefits of static type-checking[1].

[1] The need for context-dependent pretty-printers is not specific to TYPOL. It is difficult to obtain a really good pretty-printer for Ada without performing a complete type-checking.

2. Generating executable code

An initial TYPOL compiler has been designed under MENTOR. This compiler has allowed to carry out experiments, but it is still in infancy. The general strategy is to compile TYPOL to Prolog. A first compiler was written in Pascal. Then it was bootstrapped. The current compiler is written in TYPOL itself. This strategy is responsible for a very fast development, but it has resulted in many insights as well. The Prolog code that obtains is far from absurdly inefficient.

2.1. Compiling to Prolog

As it is to be expected, every rule is compiled into a Prolog clause[1]. TYPOL variables map to Prolog variables, and the denominator maps to the clause head. Each sequent maps to a *predicate*, but this mapping is performed, of course, after overloading resolution so as to distinguish predicates as much as possible at compile time. We also wish to be independent of Prolog variants that may allow more or less overloading in Prolog itself. The numerator is compiled as the body of the clause, with actions last since they are triggered *iff* the rule applies. To make certain that actions do not interfere, we see that actions should never *fail*. In fact, they should not provoke any special bindings either. The order in which the sequents and conditions on the numerator of the rule are compiled should be computed, but this is not done at the present time.

2.2. Sorting Clauses

In Prolog, the order in which clauses appear is an essential mechanism for controlling execution. If we want special case rules to have precedence over general rules, their code should be appear earlier. The constraint that compatible subjects should have least upper bounds matches Prolog very well: the compiler just needs to sort the rules topologically (or rather the clauses they generate) with respect to their subject, so that special cases occur first. The Prolog strategy of selecting the first clause that applies will then automatically choose the most specific rule.

Another optimisation may be performed by the TYPOL Compiler. Many Prolog interpreters use a double hashing strategy: first they hash on the head of the goal to find the proper collection of clauses to use, then they rehash on the head of the first argument, to try to find quickly which clause in the collection applies. To take advantage of this efficient scheme, we just need to arrange that sequents generate predicates where the subject appears as the first parameter.

[1] In fact, extensions to TYPOL that are currently examined require to generate more than one clause per rule

2.3. Pointers to source code

Tree patterns that occur in sequents are compiled into Prolog terms. When executing a specification, abstract syntax trees are mapped to Prolog terms as well. Execution of a TYPOL program takes such terms as arguments. For various reasons, we would like to refer to the source abstract tree. This is the case for example when executing the specification of a type checker: we want to see exactly where an error has been found. Another example is when executing dynamic semantics: we want to follow the execution on the source code.

To handle this problem, a pointer to the abstract syntax structure is kept in the corresponding Prolog term: if $f(X, Y)$ is an abstract syntax tree, its Prolog representation will be $f(n, X', Y')$, where X' and Y' are terms corresponding to X and Y, and n is the address of the source structure. This pointer value is just carried by Prolog, and no operations are allowed on it. This strategy makes it possible to refer to the original structure when writing actions attached to the semantic rules.

3. Examples in type-checking

The notion of types in programming languages is the object of much current research. From Algol-like languages to Ada, to languages with type-inference like ML or B, there are many examples in the literature and in computing practice exhibiting great diversity.

3.1. A very small language: PICO

The first language that we study is PICO [Pico].

The PICO type-checker on Fig. 1 illustrates the basic features of TYPOL. There are three sets of rules: the structural rules on the one hand, the rules to modify and access the environment on the other. In more complicated typing systems, we will see that a third group becomes necessary, to provide for manipulation of type expressions.

Consider first the structural rules. From rule (1), three overloadings of turnstile are introduced: to type-check a whole program, to elaborate declarations, to type-check statements.

$$\frac{\vdash \textsc{decls} : \alpha \qquad \alpha \vdash \textsc{series}}{\vdash \text{begin declare } \textsc{decls}; \textsc{series} \text{ end}} \tag{1}$$

The next three rules exhibit linear elaboration of declarations, as the type environment is built up progressively. From then on, the environment will not change anymore. Statements are checked one by one (rule (6)). Rule (7) introduces a new overloading of turnstile, to type-check expressions.

$$\frac{\alpha \vdash \textsc{id} : \tau \qquad \alpha \vdash \textsc{exp} : \tau}{\alpha \vdash \textsc{id} := \textsc{exp}} \tag{7}$$

Both sides of the assignment statement must have the same type. The only remaining non trivial rule regards the type of an identifier: to find it, the environment is looked up.

The sets **DECLARE** and **VALOF** are attempts at specifying the environment in the same style. Double declarations are not detected. On the other hand, if an identifier is undeclared, a PICO program will not type-check.

3.2. A standard example: ASPLE

This next example, somewhat more complicated than PICO, is still in the tradition of Algol-like languages, even with an Algol-68 flavour [DKL]. The rules appear on Fig. 2. Rule (4) is the first unusual rule. It introduces a new overloading for turnstile (declare a list of identifiers with a given mode). The mode with which variables are declared is not exactly the mode that appears in the program text: it is first prefixed by **ref**.

The rule for the assignment statement (rule (9)) shows that the modes of both sides need not be identical: the rule is subject to a condition. This condition is axiomatized later.

$$\frac{e \vdash \text{ID} : \mu_1 \qquad e \vdash \text{EXP} : \mu_2 \qquad LESS(\mu_1, \text{ref}\, \mu_2 :)}{e \vdash \text{ID} := \text{EXP}} \tag{9}$$

Similarly, we cannot write rules (10) (11) (12) without a condition: the language only demands that the *base* type of the expression controlling an **if** or a **while** statement be boolean. It is trivial to axiomatize this condition (see set **IS_BOOLEAN**) but we cannot use mere pattern-matching to that end.

A similar situation occurs within expressions, since **ref** operators in excess are tolerated. But they vanish from the result type of the expression. Together, the conditions (**LESS**, **IS_BOOLEAN, RESULT_TYPE**) constitute a separate package, specifically concerned with the manipulation of type expressions for ASPLE. The axiomatization that we give here shows one possible style, whose merits remain to be evaluated.

3.3. An example with overloading: KH

This language has been designed strictly for the purpose of understanding Ada-like overloading. The type-checking rules of KH are on Fig. 3. In KH, one may declare variables of an atomic type, and functions with their signatures. Then the body of a KH program is a collection of assignments, where an identifier may occur on the left, and an expression on the right hand side. However, the language exhibits *overloading* in that redeclaring an identifier with a different type is legal and meaningful. Statements are well-typed iff there is a unique way to type them.

The basic idea in the type-checker is to write the rules *as if* overloading did not exist, and then, instead of considering the environment as a function, make it a **relation**. The backtracking inherent in Prolog will suffice to resolve overloading.

Remark: the algorithm that results is not inefficient at all, because the environment is searched with a pair consisting of an identifier plus a partially instantiated type, due to rule (13) and (14).

$$\frac{\rho \vdash \text{ID} : \tau \qquad \rho \vdash \text{EXP} : \tau}{\rho \vdash \text{ID} := \text{EXP}} \tag{13}$$

$$\frac{\rho \vdash \text{ID} : (\tau_1) \rightarrow \tau \qquad \rho \vdash \text{LEXP} : \tau_1}{\rho \vdash \text{ID}(\text{LEXP}) : \tau} \tag{14}$$

We see here that unification is an interesting way to accumulate constraints on types.

One interesting application of KH is in typechecking abstract syntax definitions. An abstract syntax will be naturally presented as a KH declarative part. Overloading is mandatory to avoid heavy coercions everywhere. Now checking whether a given term built with abstract syntax constructors is legal is just type-checking that term with respect to this declarative part. Note that in fact, any two operators with the same name will always be declared with the same type of *arguments*, and only differ by their result-type. As a consequence, resolving overloading will involve no backtracking, and it will be very incremental.

The need for this type-checking algorithm occurs immediately in our context, to check that the abstract syntax terms occuring in a TYPOL program really denote existing subtrees in their language.

Understanding overloading in that manner seems to clarify the distinction between overloading and polymorphism: *overloading* is a property that comes from the definition of the *environment* of a language. On the other hand *polymorphism* is a property of the *type expressions*, that may contain free variables.

3.4. An example with type inference: mini-ML

To assess the robustness of our notation, following [TD] we are now attempting a more ambitious task: to provide a specification of a central part of the ML language [ML]. ML is a language with two very interesting characteristics from the point of view of types: *polymorphism* and *type inference*. ML typechecking is the object of numerous discussions in the literature ([DM],[Cardelli], [Reynolds]). As such, it is a natural challenge for TYPOL. The features of ML retained in this mini language include all functional aspects plus cartesian products but no lists nor abstract types.

Consider the first set of rules, the set **TYPE**, which is the core of the type-checking specification. Rules (1),(2) and (3) are without mystery. Rule (10) concerns cartesian products and is easy as well. Rule (9) on conditional expressions is just what one would expect to write, but it already implies some form of unification since there are repeated occurrences of the same type variable τ. Rule (7) just tells that a let construct reduces to an application whose operand is an explicit lambda-expression, a case handled by rule (5).

Rule (4) shows that the language has type-inference, since the variable (or list of variables) P is declared with a type τ that is a *free* variable, to be later constrained by further unifications.

$$\frac{\pi \vdash \text{P}, \tau : \pi_1 \qquad \pi_1 \vdash \text{E} : \tau_1}{\pi \vdash \lambda \text{P.E} : \tau \rightarrow \tau_1} \tag{4}$$

This means that the type of the lambda-expression will contain *free variables*. Rules (5) and (6) are really at the core of polymorphism. Rule (5) is a special case of rule (6) hence it is preferred when applicable. Rule (6) is another fancy use of unification and is a sort of *modus ponens* for types.

Now rule (5) deserves more analysis.

$$\frac{\pi \vdash \text{E}_2 : \tau_2 \qquad \pi \overset{\text{gen}}{\vdash} \tau_2 : \tau \qquad \pi \vdash \text{P}, \tau : \pi_1 \qquad \pi_1 \vdash \text{E}_1 : \tau_1}{\pi \vdash (\lambda \text{P.E}_1)\text{E}_2 : \tau_1} \tag{5}$$

In the case where the operator of an application is an explicit lambda expression, we can rely on the information gathered on the operand to help in type-checking the operator. The variable (or variables) bound by λ are declared of a type which *generalises* that of the operand, with respect to the variables that remain free in the environment. This is done in rule set **GEN**. Generalising a type-expression merely consists in tagging its variables with the *gen* constructor whenever they occur free in the environment.

Of course, this generalisation cannot be understood without looking at how the types of identifiers are found in the environment. This is explained in rule (11). Type values found in the environment may contain tagged variables. Whenever we fetch a type value in the environment, all of its tagged variables are consistently made into new variables. This is axiomatized in rule set **RENAME**.

The management of the type environment in Mini-ML is trivial, but we see that we need two primitives to manipulate type-expressions. Type expressions may have to be *generalized*, and they may have to be *renamed*. The set **FREEVARS** is just an auxiliary collection of rules to compute the free variables in an environment[1].

Type-checking of Mini-ML is difficult and rich in lessons for our formalism. While we succeed in a very dense description, the need to rename a type-expression is probably linked to some awkwardness in handling quantifiers within type expressions. This is an area where further study is needed. Still, we have successfully pushed unification into the *meta-system*, as well as all generation of new identifiers.

3.5. Further examples in typechecking

As already noted, the area of types is a very important subject of current research in programming languages. We have not dealt with recursive data types or types where quantifiers occur in a deeper way than in ML. Nor did we look at modules [DMQ]. Finally, ideas linked to type inclusion, such as coercions and inheritance, must also be investigated.

More complex forms of overloading, and in general more complex environment structures must be studied as well, to gain understanding in what formalism is best suited to describe them.

[1] See an interesting and intuitive use of overloading in that set

4. Examples in translation

Translation from one language to another is in general heavily guided by the structure of the source formalism. Hence it should not come as a surprise that TYPOL makes specifying translations easy.

4.1. From surface abstract syntax to deep abstract syntax

Defining an abstract syntax for a language that has not been designed with one in mind is not very easy. The designer is always oscillating between two contradictory goals:

- Build a natural abstract syntax from the point of view of structured editing, and program transformations.
- Stay away from the vagaries of concrete syntax and bring some order to the language, grouping semantic concepts under the same operators, performing normalizations, etc... In this way the language's semantics will be simpler.

In a well designed language like ML, the abstract syntax put forward by the language's designers is not adequate for structure editing, because it is too *deep*, it has lost too much superficial information that is necessary to reconstruct a reasonably looking program. This is felt in particular in the area of error messages: diagnostic directed to the user are couched in terms of the deep abstract syntax, a syntax that the user doesn't know.

We have come to the conclusion that this dilemma will *not* go away, and that we must deal with (at least) two abstract syntaxes: the *superficial* abstract syntax will be adequate for structure editing and allow reconstructing the entire program text up to some innocuous normalizations. Some constructors will disappear from the *deep abstract syntax*, as a result of *canonicalization*. But some new constructors will also appear as a result of overloading resolution. We will always specify dynamic semantics on the deep abstract syntax, and static semantics on the superficial abstract syntax. Translation from superficial to deep abstract syntax follows usually the rules for type-checking and presents no additional difficulties. We prefer to discuss translation on examples where the source and target language are very different.

4.2. Generating code for a stack machine (SML)

We take as a first example the translation from ASPLE (deep abstract syntax) to a simple stack machine language (SML) whose semantics is discussed in detail later. The definition of the translation appears on Fig. 5. Rule (1) shows that elaboration of declarations produces a store, while translation of statements produces some code. Both components are returned as result of the translation. Creation of the store (rules (2) to (6)) is not very exciting. Rule (7) and (8) show that code for successive statements is simply concatenated. The intuitive ideas about how to generate code are mapped directly to the next rules. Notice that the **block** instruction limits the scope of labels so that we never use any other label than 1 or 2. If we had to use arbitrarily many, we could have relied on our mechanism to introduce free variables to generate labels. With some care, we could even generate directly relocatable code.

This translation exercise is a pure homomorphism, hence it does not use much of the real power of TYPOL.

4.3. Generating code for a functional language

Recently, G. Cousineau and P-L. Curien have proposed a very ingenious abstract machine for the compilation of ML [CAM]. The complete semantics of the machine is described later. We present on Fig. 6 the translation from Mini-ML to CAM.

The first interesting rule is rule (9) where we see some use of pattern matching. One can see on that rule that only *functions* are allowed to be defined recursively in Mini-ML, a fact that the type-checker will have verified. Aside from that, rule (8) and (9) have the same outlook. Rule (11) shows an optimisation of rule (13) for predefined functions, rule (12) shows another optimisation in the case where an explicit lambda-expression occurs as the operator of an application.

We see in this example that the principle of selecting the most specific rule gives us a form of optimisation for free.

4.4. A strategy for pretty-printers

Pretty-printers produce text from abstract syntax trees. Most existing pretty-printers have two defects in our eyes:

- They are specifically line oriented
- Their specification contains information that pertains to the output device.

We believe that a two-pass strategy is more reasonable: the specification of a pretty-printer should just be a translation from source language to a universal *formatting language*. This language is universal in the sense that it is independent of the particular language that one wants to pretty-print, but also because it can be executed on a variety of output devices with results that are more or less refined. The INTERSCRIPT proposal [Jol] is a language of that kind, although possibly too ambitious for our purposes.

In that perspective, pretty-printing becomes just a particular kind of translation, which should not be in general very hard to specify. Fine tuning of a pretty-printer, however, will make extensive use of our facility for special cases.

5. Examples in specifying abstract machines

5.1. Specifying SML

The specification of the semantics of SML is given on Fig. 7. This specification illustrates several interesting traits of TYPOL:

- The state of the machine is a pair, represented between angle brackets, of a store and a stack. All stacking and unstacking appears merely through pattern matches. See for example rule (17) for the **sto** instruction

$$\frac{r \overset{\texttt{update}}{\vdash} x, \varphi : r_1}{\rho \vdash \mathrm{sto}, <r, \varphi \cdot x \cdot k> : <r_1, k>} \qquad (17)$$

  or rule (13) for the **ldo** instruction. The store is accessed through two primitives, **get** and **update**, and these primitives are candidate for implementation in a different style. Given this, most of the rules are axioms, as it is to be expected in the description of a virtual machine.
- The only non-trivial part in this example is the treatment of jumps. In SML, jumps are local to a block, signaled by a **block** instruction. Hence an environment, mapping labels to "continuations" is built upon entry in the block. This style is of course heavily influenced by denotational semantics. Then when a jump instruction has to be executed, the continuation is fetched in the environment, and execution proceeds with that continuation. The best example is provided by the unconditional jump instruction, **ujp**, in rule (3).

$$\frac{\rho \overset{\texttt{cont\_find}}{\vdash} \mathrm{lbl}\,\mathrm{L} : c_1 \qquad \rho \vdash c_1, s : s_1}{\rho \vdash \mathrm{coms}[\mathrm{ujp}\,\mathrm{L} \cdot c], s : s_1} \qquad (3)$$

Now, in order not to deal with continuations in instructions that do not alter control flow, we set up a general rule to describe sequences of instructions, rule (8).

$$\frac{\rho \vdash \mathrm{COM}, s : s_1 \qquad \rho \vdash \mathrm{COMS}, s_1 : s_2}{\rho \vdash \mathrm{coms}[\mathrm{COM} \cdot \mathrm{COMS}], s : s_2} \qquad (8)$$

Then, all descriptions of jump instructions will be given in rules whose denominator is a special case of that general rule, and hence supersede it.

This effort at having both a *direct* and a *continuation* semantics would become useless if all instructions could provoke a transfer of control, for example by raising an exception. But as it stands, we feel that this example exhibits once again the usefulness of our style that allows for special cases.

5.2. Specifying the Categorical Abstract Machine (CAM)

The complete specification of the machine CAM is shown on Fig. 8. The simplicity and elegance of the machine is evident from the specification. The state of the machine is again displayed with angle brackets, and can be thought of either as a register-stack pair, or as a single stack. Pattern-matching suffices to explain the simple instructions: **car**, **cdr**, **cons**, **push**, **swap**. Notice that it is used not only to match the stack but also to match stack values. There are no jumps, only closures and recursive closures, so that the sequencing rule has no exception. Conditional branching is simple as well.

The **cur** and **recf** instruction are completely analogous. It is only the semantics of apply (the **app** instruction) that will show a difference between closures and recursive closures. There are two rules regarding **app**. Which one should be used depends on what is on top of the stack. Both rules extract the code part and they differ only in what environment this code is executed in. For a closure, the environment of the closure is prefixed to the current environment. For a recclosure, the current environment is prefixed with both the recclosure's own environment, and the recclosure itself.

As a matter of interest, we show an alternate rule for the **recf** instruction on Fig 9. This rule is intuitive and simple, and permits to dispense with recclosure altogether. It shows even more clearly than before that a knot is being tied in the environment. But it introduces an infinite term in our definition, something we would like to avoid for the moment.

6. Examples in specifying Dynamic Semantics

Experiments with dynamic semantics are only at a beginning. This is an area of excellence for Denotational Semantics, so we must convince ourselves that the new style is indeed clearer and easier.

6.1. ASPLE

The dynamic semantics of ASPLE is extremely simple. It is specified on Fig. 10. Let us make a few remarks, pointing out how to read the TYPOL notation. Rule (1) tells to start from an empty store, performing declarations then statements. Rules (2) to (6) show that uninitialized store is being allocated for all identifiers in the program, regardless of their declared mode. From rule (1) we see that turnstile is overloaded in three ways (execute a whole program, elaborate declarations, execute statements). Now in rule (9), defining the assignment statement, we learn of a fourth overloading, to evaluate expressions. The next interesting rule is rule (14) where we see how simply a **while** statement is handled. The **deref** constructor causes no extra difficulty, because most of the work is done at type-checking time for ASPLE.

6.2. Mini-ML

The specification on Fig. 11 is a first attempt at specifying the dynamic semantics of our applicative language, Mini-ML. Since the only semantic function to define is how to evaluate an expression, the turnstile symbol is not overloaded here. The environment is treated in a slightly different style. Instead of making a separate set of rules to explain how to *add* bindings in the environment, we perform these additions directly in the rules. But we still keep a separate set of rules to describe how to *retrieve* bindings in the environment. This different style seems clearer in the particular case of Mini-ML, and more generally in dynamic semantics. In rule (9) regarding the **let** construct (the same rule as rule (13)) we see how a new binding is simply prefixed to the environment. Rule (14) shows how to evaluate an application, when the operator of the application is the result of an evaluation.

$$\frac{\rho \vdash \mathrm{E}_2 : \alpha \qquad \rho \vdash \mathrm{E}_1 : \mathit{closure}(\lambda \mathrm{P.E}, \rho_1) \qquad \rho_2 = \mathit{env}[\mathrm{P} \mapsto \alpha; \rho_1; \rho] \qquad \rho_2 \vdash \mathrm{E} : \beta}{\rho \vdash \mathrm{E}_1\,\mathrm{E}_2 : \beta} \quad (14)$$

The second sequent on the numerator uses pattern-matching to obtain three components: the body of the closure, the name of the bound variable and the closure's environment.

It is interesting to observe that specifying the dynamic semantics of Mini-ML is substantially simpler than specifying its type-checking rules.

7. Remarks on some difficulties

In the course of the discussion of the examples above, we have seen that several difficulties remain:

- Our formalism supposes that we have a Prolog with occur-check, or at least that we can turn on occur-check on some rules. For example in Mini-ML type-checking $\lambda x.xx$ without occur-check will loop instead of failing.
- Our formalism seems somewhat weak when dealing with values that may contain binders.
- We need to have more experiments on how to deal with failure: failure to type-check, failure to execute. This is mostly a pragmatic issue.
- Sometimes we want a functional environment, sometimes we want it to be a relation. How do we decide whether to generate *cuts* or not for the structural rules?
- It is clear that Prolog is a good target language from the point of view of *speed*. Shall we run into problems of *storage* with large examples?
- Much more rule analysis is needed to generate good code on the one hand, but also to be ready for incremental evaluation mechanisms.

ACKNOWLEDGMENTS

G. Plotkin gave the initial impetus for this work during a meeting in Fontainebleau. The skepticism of D. MacQueen served as a healthy challenge. G. Cousineau and G. Huet helped us to figure out ML typechecking, the CAM machine, and the structure of sequents. Thanks go to J. Reynolds and G. Lindstrom for many discussions in Sophia-Antipolis.

REFERENCES

[Plotkin] Plotkin, G.D., "A Structural Approach to Operational Semantics", DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[DKL] V. Donzeau-Gouge, G. Kahn, B. Lang, "A complete machine checked definition of a simple programming language using denotational semantics", INRIA Research Report 330, October 1978.

[ML] M. Gordon, R. Milner, C. Wadsworth, G. Cousineau, G.Huet, L. Paulson, "The ML Handbook, Version 5.1", INRIA, October 1984

[DM] L. Damas, R. Milner, "Principal type-schemes for functional programs", Proceedings POPL 1982, pp.207-212.

[Cardelli] L. Cardelli, "Basic Polymorphic Type-checking", Polymorphism, January 1985

[Reynolds] J.C. Reynolds, "Three Approaches to Type Structure", Proceedings TAPSOFT, Lecture Notes in Computer Science, Vol. 185, March 1985

[CProlog] F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira, "CProlog User's Manual, Version 1.2" EdCAAD, Department of Architecture, University of Edinburgh, U.K.(1983)

[CAM] G. Cousineau, P. L. Curien, M. Mauny, "The Categorical Abstract Machine", Report 85-8, LITP, University Paris VII, January 1985

[TD] Th. Despeyroux, "Executable Specification of Static Semantics", in Semantics of Data Types, Lecture Notes in Computer Science, Vol. 173, June 1984

[DMQ] D. B. MacQueen, "Modules for standard ML", Private Communication, 1984

[Mentor] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, "Programming environments based on structured editors: The Mentor experience" INRIA Research Report no. 26, July 1980

[Pico] J.A. Bergstra, J. Heering. P. Klint "Algebraic definition of a simple programming language", CWI Report CS-R8504, February 1985

[Jol] V. Joloboff, R. Pierce, T. Schleich, "INTERSCRIPT", ISO /TC97/SC18/WG3/M439, 1985

Figure 1

program PICO_TC is
use PICO
$\alpha, \alpha_1, \alpha_2 : \mathit{TENV}$;
$\tau : \mathit{TYPE}$;

$$\frac{\vdash \mathrm{DECLS} : \alpha \qquad \alpha \vdash \mathrm{SERIES}}{\vdash \mathbf{begin\ declare}\ \mathrm{DECLS}; \mathrm{SERIES}\ \mathbf{end}} \tag{1}$$

$$\frac{\mathit{env}[\,] \vdash \mathrm{DECLS} : \alpha}{\vdash \mathrm{DECLS} : \alpha} \tag{2}$$

$$\alpha \vdash \mathbf{decls}[\,] : \alpha \tag{3}$$

$$\frac{\alpha \overset{\mathbf{declare}}{\vdash} \mathrm{X}, \mathrm{T} : \alpha_1 \qquad \alpha_1 \vdash \mathrm{DECLS} : \alpha_2}{\alpha \vdash \mathrm{X} : \mathrm{T}, \mathrm{DECLS} : \alpha_2} \tag{4}$$

$$\alpha \vdash \mathbf{series}[\,] \tag{5}$$

$$\frac{\alpha \vdash \mathrm{STAT} \qquad \alpha \vdash \mathrm{SERIES}}{\alpha \vdash \mathrm{STAT}; \mathrm{SERIES}} \tag{6}$$

$$\frac{\alpha \vdash \mathrm{ID} : \tau \qquad \alpha \vdash \mathrm{EXP} : \tau}{\alpha \vdash \mathrm{ID} := \mathrm{EXP}} \tag{7}$$

$$\frac{\alpha \vdash \mathrm{EXP} : \mathbf{int} \qquad \alpha \vdash \mathrm{SERIES}_1 \qquad \alpha \vdash \mathrm{SERIES}_2}{\alpha \vdash \mathbf{if}\ \mathrm{EXP}\ \mathbf{then}\ \mathrm{SERIES}_1\ \mathbf{else}\ \mathrm{SERIES}_2\ \mathbf{fi}} \tag{8}$$

$$\frac{\alpha \vdash \mathrm{EXP} : \mathbf{int} \qquad \alpha \vdash \mathrm{SERIES}}{\alpha \vdash \mathbf{while}\ \mathrm{EXP}\ \mathbf{do}\ \mathrm{SERIES}\ \mathbf{od}} \tag{9}$$

$$\frac{\alpha \overset{\mathbf{valof}}{\vdash} \mathbf{id}\ \mathrm{X} : \tau}{\alpha \vdash \mathbf{id}\ \mathrm{X} : \tau} \tag{10}$$

$$\alpha \vdash \mathbf{number}\ \mathrm{N} : \mathbf{int} \tag{11}$$

$$\alpha \vdash \mathbf{string}\ \mathrm{S} : \mathbf{str} \tag{12}$$

$$\frac{\alpha \vdash \mathrm{EXP}_1 : \mathbf{int} \qquad \alpha \vdash \mathrm{EXP}_2 : \mathbf{int}}{\alpha \vdash \mathrm{EXP}_1 + \mathrm{EXP}_2 : \mathbf{int}} \tag{13}$$

$$\frac{\alpha \vdash \mathrm{EXP}_1 : \mathbf{str} \qquad \alpha \vdash \mathrm{EXP}_2 : \mathbf{str}}{\alpha \vdash \mathrm{EXP}_1 \| \mathrm{EXP}_2 : \mathbf{str}} \tag{14}$$

set DECLARE is

$$\mathit{env}[\,] \vdash \mathrm{X}, \tau : \mathit{env}[\mathrm{X} \mapsto \tau] \tag{1}$$

$$\mathit{env}[\mathrm{X} \mapsto \tau_1 \cdot \mathrm{ENV}] \vdash \mathrm{X}, \tau : \mathit{env}[\mathrm{X} \mapsto \tau \cdot \mathrm{ENV}] \tag{2}$$

$$\frac{\mathrm{ENV} \vdash \mathrm{X}, \tau : \mathrm{ENV}_1}{\mathit{env}[\mathrm{T} \cdot \mathrm{ENV}] \vdash \mathrm{X}, \tau : \mathit{env}[\mathrm{T} \cdot \mathrm{ENV}_1]} \tag{3}$$

end DECLARE ;

Figure 1

set VALOF is

$$env[\mathrm{X} \mapsto \tau \cdot \mathrm{ENV}] \vdash \mathrm{X} : \tau \tag{1}$$

$$\frac{\mathrm{ENV} \vdash \mathrm{X} : \tau}{env[\mathrm{PAIR} \cdot \mathrm{ENV}] \vdash \mathrm{X} : \tau} \tag{2}$$

end VALOF ;

end PICO_TC

Figure 2

program ASPLE_NTC is
use ASPLE
$e, e_0, e_1, e_2, e_3 : \mathit{ASPLE\_ENV};$
$\mu, \mu_1, \mu_2, \mu_3 : \mathit{ASPLE};$

$$\frac{\mathbf{env}[\,] \vdash \mathrm{DECLS} : e \qquad e \vdash \mathrm{STMS}}{\vdash \mathbf{begin}\ \mathrm{DECLS}\ \mathrm{STMS}\ \mathbf{end}} \tag{1}$$

$$\frac{e \vdash \mathrm{DECL} : e_1}{e \vdash \mathrm{DECL} : e_1} \tag{2}$$

$$\frac{e \vdash \mathrm{DECL} : e_1 \qquad e_1 \vdash \mathrm{DECLS} : e_2}{e \vdash \mathrm{DECL};\mathrm{DECLS} : e_2} \tag{3}$$

$$\frac{e \vdash \mathrm{IDLIST}, \mathbf{ref}\ \mathrm{MODE} : e_1}{e \vdash \mathrm{MODE}\ \mathrm{IDLIST} : e_1} \tag{4}$$

$$\frac{e \overset{\mathbf{declare}}{\vdash} \mathbf{id}\ \mathrm{X}, \mu : e_1}{e \vdash \mathbf{id}\ \mathrm{X}, \mu : e_1} \tag{5}$$

$$\frac{e \overset{\mathbf{declare}}{\vdash} \mathbf{id}\ \mathrm{X}, \mu : e_1 \qquad e_1 \vdash \mathrm{IDLIST}, \mu : e_2}{e \vdash \mathbf{id}\ \mathrm{X}, \mathrm{IDLIST}, \mu : e_2} \tag{6}$$

$$\frac{e \vdash \mathrm{STM}}{e \vdash \mathrm{STM}} \tag{7}$$

$$\frac{e \vdash \mathrm{STM} \qquad e \vdash \mathrm{STMS}}{e \vdash \mathrm{STM};\mathrm{STMS}} \tag{8}$$

$$\frac{e \vdash \mathrm{ID} : \mu_1 \qquad e \vdash \mathrm{EXP} : \mu_2 \qquad \mathit{LESS}(\mu_1, \mathbf{ref}\ \mu_2 :)}{e \vdash \mathrm{ID} := \mathrm{EXP}} \tag{9}$$

$$\frac{e \vdash \mathrm{EXP} : \mu \qquad \mathit{IS\_BOOLEAN}(\mu :) \qquad e \vdash \mathrm{STMS}}{e \vdash \mathbf{if}\ \mathrm{EXP}\ \mathbf{then}\ \mathrm{STMS}\ \mathbf{fi}} \tag{10}$$

$$\frac{e \vdash \mathrm{EXP} : \mu \qquad \mathit{IS\_BOOLEAN}(\mu :) \qquad e \vdash \mathrm{STMS}_1 \qquad e \vdash \mathrm{STMS}_2}{e \vdash \mathbf{if}\ \mathrm{EXP}\ \mathbf{then}\ \mathrm{STMS}_1\ \mathbf{else}\ \mathrm{STMS}_2\ \mathbf{fi}} \tag{11}$$

$$\frac{e \vdash \mathrm{EXP} : \mu \qquad \mathit{IS\_BOOLEAN}(\mu :) \qquad e \vdash \mathrm{STMS}}{e \vdash \mathbf{while}\ \mathrm{EXP}\ \mathbf{do}\ \mathrm{STMS}\ \mathbf{end}} \tag{12}$$

$$\frac{e \vdash \mathrm{ID} : \mu}{e \vdash \mathbf{input}\ \mathrm{ID}} \tag{13}$$

$$\frac{e \vdash \mathrm{EXP} : \mu}{e \vdash \mathbf{output}\ \mathrm{EXP}} \tag{14}$$

$$\frac{e \overset{\mathbf{type\_of}}{\vdash} \mathbf{id}\ \mathrm{X} : \mu}{e \vdash \mathbf{id}\ \mathrm{X} : \mu} \tag{15}$$

$$e \vdash \mathbf{boolean}\ \mathrm{X} : \mathbf{bool} \tag{16}$$

$$e \vdash \mathbf{number}\ \mathrm{X} : \mathbf{int} \tag{17}$$

Figure 2

$$\frac{e \vdash \mathrm{EXP}_1 : \mu_1 \qquad e \vdash \mathrm{EXP}_2 : \mu_2 \qquad \overset{\mathbf{result\_type}}{\vdash} \mu_1, \mu_2 : \mu_3}{e \vdash \mathrm{EXP}_1 \neq \mathrm{EXP}_2 : \mathrm{bool}} \tag{18}$$

$$\frac{e \vdash \mathrm{EXP}_1 : \mu_1 \qquad e \vdash \mathrm{EXP}_2 : \mu_2 \qquad \overset{\mathbf{result\_type}}{\vdash} \mu_1, \mu_2 : \mu_3}{e \vdash \mathrm{EXP}_1 = \mathrm{EXP}_2 : \mathrm{bool}} \tag{19}$$

$$\frac{e \vdash \mathrm{EXP}_1 : \mu_1 \qquad e \vdash \mathrm{EXP}_2 : \mu_2 \qquad \overset{\mathbf{result\_type}}{\vdash} \mu_1, \mu_2 : \mu}{e \vdash \mathrm{EXP}_1 \ \mathrm{OP}\ \mathrm{EXP}_2 : \mu} \tag{20}$$

set DECLARE is

$$env[\,] \vdash \mathrm{id}\,\mathrm{X}, \mu : env[\mathrm{id}\,\mathrm{X} \mapsto \mu] \tag{1}$$

$$\frac{\mathrm{X} \neq \mathrm{Y} \qquad \mathrm{E} \vdash \mathrm{id}\,\mathrm{X}, \mu : \mathrm{L}_1}{env[\mathrm{id}\,\mathrm{Y} \mapsto \mu_1 \cdot \mathrm{E}] \vdash \mathrm{id}\,\mathrm{X}, \mu : env[\mathrm{id}\,\mathrm{Y} \mapsto \mu_1 \cdot \mathrm{L}_1]} \tag{2}$$

end DECLARE ;

set TYPE_OF is

$$env[\mathrm{id}\,\mathrm{X} \mapsto \mu \cdot \mathrm{E}] \vdash \mathrm{id}\,\mathrm{X} : \mu \tag{2}$$

$$\frac{\mathrm{E} \vdash \mathrm{id}\,\mathrm{X} : \mu}{env[\mathrm{TYPE} \cdot \mathrm{E}] \vdash \mathrm{id}\,\mathrm{X} : \mu} \tag{3}$$

end TYPE_OF ;

set LESS is

$$\vdash \mu, \mu \tag{1}$$

$$\frac{\vdash \mu, \mathrm{M}_1}{\vdash \mu, \mathrm{ref}\,\mathrm{M}_1} \tag{2}$$

end LESS ;

set IS_BOOLEAN is

$$\vdash \mathrm{bool} \tag{1}$$

$$\frac{\vdash \mathrm{M}}{\vdash \mathrm{ref}\,\mathrm{M}} \tag{2}$$

end IS_BOOLEAN ;

Figure 2

set RESULT_TYPE is

$$\frac{\vdash M_1, M_2 : \mu}{\vdash \text{ref}\, M_1, \text{ref}\, M_2 : \mu} \quad (1)$$

$$\frac{\vdash \mu, M : \mu}{\vdash \mu, \text{ref}\, M : \mu} \quad (2)$$

$$\frac{\vdash M, \mu : \mu}{\vdash \text{ref}\, M, \mu : \mu} \quad (3)$$

$$\vdash \mu, \mu : \mu \quad (4)$$

end RESULT_TYPE ;

end ASPLE_NTC

Figure 3

program KH_TC is
use KH
$\rho, \rho_1, \rho_2 : \mathit{ENV};$
$\tau, \tau_1, \tau_2, \tau_3, \tau_4 : \mathit{TYPE};$

$$\frac{\mathit{env}[\,] \vdash \text{DECLS} : \rho \qquad \rho \vdash \text{LSTATS}}{\vdash \textbf{declare}\,\text{DECLS}\,\textbf{in}\,\text{LSTATS}} \tag{1}$$

$$\rho \vdash \textbf{decls}[\,] : \rho \tag{2}$$

$$\frac{\rho \vdash \text{DECL} : \rho_1 \qquad \rho_1 \vdash \text{DECLS} : \rho_2}{\rho \vdash \text{DECL};\text{DECLS} : \rho_2} \tag{3}$$

$$\frac{\vdash \text{TYPE} : \tau \qquad \rho \overset{\text{declare}}{\vdash} \text{ID}, \tau : \rho_1}{\rho \vdash \text{ID} : \text{TYPE} : \rho_1} \tag{4}$$

$$\vdash \textbf{id}\,\text{X} : \textbf{id}\,\text{X} \tag{5}$$

$$\vdash \textbf{type}\,\text{X} : \textbf{type}\,\text{X} \tag{6}$$

$$\frac{\vdash \tau_1 : \tau_2 \qquad \vdash \tau_3 : \tau_4}{\vdash (\tau_1) \rightarrow \tau_3 : (\tau_2) \rightarrow \tau_4} \tag{7}$$

$$\frac{\vdash \tau_1 : \tau_2}{\vdash \textbf{list}(\tau_1) : \textbf{list}(\tau_2)} \tag{8}$$

$$\vdash \textbf{ltypes}[\,] : \textbf{ltypes}[\,] \tag{9}$$

$$\frac{\vdash \text{STYPE} : \tau \qquad \vdash \text{LTYPES} : \tau_1}{\vdash \text{STYPE}, \text{LTYPES} : \tau, \tau_1} \tag{10}$$

$$\rho \vdash \textbf{lstats}[\,] \tag{11}$$

$$\frac{\rho \vdash \text{STAT} \qquad \rho \vdash \text{STATS}}{\rho \vdash \text{STAT};\text{STATS}} \tag{12}$$

$$\frac{\rho \vdash \text{ID} : \tau \qquad \rho \vdash \text{EXP} : \tau}{\rho \vdash \text{ID} := \text{EXP}} \tag{13}$$

$$\frac{\rho \vdash \text{ID} : (\tau_1) \rightarrow \tau \qquad \rho \vdash \text{LEXP} : \tau_1}{\rho \vdash \text{ID}(\text{LEXP}) : \tau} \tag{14}$$

$$\frac{\rho \vdash \text{EXP} : \tau \qquad \rho \vdash \text{LEXP} : \tau_1}{\rho \vdash \text{EXP}, \text{LEXP} : \tau, \tau_1} \tag{15}$$

$$\rho \vdash \textbf{lexps}[\,] : \textbf{ltypes}[\,] \tag{16}$$

$$\frac{\rho \overset{\text{typeof}}{\vdash} \textbf{id}\,\text{X} : \tau}{\rho \vdash \textbf{id}\,\text{X} : \tau} \tag{17}$$

set DECLARE is

$$\rho \vdash \text{ID}, \tau : \mathit{env}[\text{ID} \mapsto \tau \cdot \rho] \tag{1}$$

end DECLARE ;

Figure 3

set TYPEOF is

$$env[\mathrm{ID} \mapsto \tau \cdot \rho] \vdash \mathrm{ID} : \tau \qquad (1)$$

$$\frac{\rho \vdash \mathrm{ID} : \tau}{env[\mathrm{PAIR} \cdot \rho] \vdash \mathrm{ID} : \tau} \qquad (2)$$

end TYPEOF ;

end KH_TC

Figure 4

program L_TC **is**
use L
l_exp : *L*;
π_0, π, π_1, π_2 : *TENV*;
$\tau_0, \tau, \tau_1, \tau_2, \tau_3, \tau_4$: *TYPES*;
l, l_1, l_2 : *LIST*;

$$\frac{\pi_0 = \mathit{init\_env}() \qquad \pi_0 \overset{\text{type}}{\vdash} \mathit{l\_exp} : \tau \qquad \mathit{PRINT\_TYPE}(\tau\,;)}{\vdash \mathit{l\_exp}} \tag{1}$$

set TYPE is

$$\pi \vdash \text{true} : \mathit{bool} \tag{1}$$

$$\pi \vdash \text{false} : \mathit{bool} \tag{2}$$

$$\pi \vdash \text{number } \mathrm{N} : \mathit{int} \tag{3}$$

$$\frac{\pi \vdash \mathrm{P}, \tau : \pi_1 \qquad \pi_1 \vdash \mathrm{E} : \tau_1}{\pi \vdash \lambda \mathrm{P}.\mathrm{E} : \tau \rightarrow \tau_1} \tag{4}$$

$$\frac{\pi \vdash \mathrm{E}_2 : \tau_2 \qquad \pi \overset{\text{gen}}{\vdash} \tau_2 : \tau \qquad \pi \vdash \mathrm{P}, \tau : \pi_1 \qquad \pi_1 \vdash \mathrm{E}_1 : \tau_1}{\pi \vdash (\lambda \mathrm{P}.\mathrm{E}_1)\mathrm{E}_2 : \tau_1} \tag{5}$$

$$\frac{\pi \vdash \mathrm{E}_2 : \tau_2 \qquad \pi \vdash \mathrm{E}_1 : \tau_2 \rightarrow \tau}{\pi \vdash \mathrm{E}_1\ \mathrm{E}_2 : \tau} \tag{6}$$

$$\frac{\pi \vdash (\lambda \mathrm{P}.\mathrm{E}_1)\mathrm{E}_2 : \tau}{\pi \vdash \text{let } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1 : \tau} \tag{7}$$

$$\frac{\pi \vdash (\lambda \mathrm{P}.\mathrm{E}_2)(\text{fix } \lambda \mathrm{P}.\mathrm{E}_1) : \tau}{\pi \vdash \text{letrec } \mathrm{P} = \mathrm{E}_1 \text{ in } \mathrm{E}_2 : \tau} \tag{8}$$

$$\frac{\pi \vdash \mathrm{E}_1 : \mathit{bool} \qquad \pi \vdash \mathrm{E}_2 : \tau \qquad \pi \vdash \mathrm{E}_3 : \tau}{\pi \vdash \text{if } \mathrm{E}_1 \text{ then } \mathrm{E}_2 \text{ else } \mathrm{E}_3 : \tau} \tag{9}$$

$$\frac{\pi \vdash \mathrm{E}_1 : \tau_1 \qquad \pi \vdash \mathrm{E}_2 : \tau_2}{\pi \vdash (\mathrm{E}_1, \mathrm{E}_2) : \tau_1 \times \tau_2} \tag{10}$$

$$\frac{\pi \overset{\text{typeof}}{\vdash} \text{ident } \mathrm{X} : \tau_0 \qquad \overset{\text{rename}}{\vdash} \tau_0 : \tau}{\pi \vdash \text{ident } \mathrm{X} : \tau} \tag{11}$$

$$\frac{\pi \vdash \mathrm{P}_1, \tau_1 : \pi_1 \qquad \pi_1 \vdash \mathrm{P}_2, \tau_2 : \pi_2}{\pi \vdash (\mathrm{P}_1, \mathrm{P}_2), \tau_1 \times \tau_2 : \pi_2} \tag{12}$$

$$\frac{\pi \overset{\text{declare}}{\vdash} \text{ident } \mathrm{X}, \tau : \pi_1}{\pi \vdash \text{ident } \mathrm{X}, \tau : \pi_1} \tag{13}$$

end TYPE ;

Figure 4

set DECLARE is

$$\text{DECLS} \vdash \text{X}, \tau : decls[\text{X} \mapsto \tau \cdot \text{DECLS}] \tag{1}$$

end DECLARE ;

set TYPEOF is

$$decls[\text{X} \mapsto \tau \cdot \text{DECLS}] \vdash \text{X} : \tau \tag{1}$$

$$\frac{\text{DECLS} \vdash \text{X} : \tau}{decls[\text{DECL} \cdot \text{DECLS}] \vdash \text{X} : \tau} \tag{2}$$

end TYPEOF ;

set FREEVARS is

$$\frac{\pi \vdash list\_var[] : l}{\vdash \pi : l} \tag{1}$$

$$\frac{\tau \vdash list\_var[] : l}{\vdash \tau : l} \tag{2}$$

$$\frac{ISVAR(\tau :) \qquad ISIN(l, \tau :)}{l \vdash \tau : l} \tag{3}$$

$$\frac{ISVAR(\tau :)}{l \vdash \tau : list\_var[\tau \cdot l]} \tag{4}$$

$$l \vdash int : l \tag{5}$$

$$l \vdash bool : l \tag{6}$$

$$l \vdash gen(\tau) : l \tag{7}$$

$$\frac{l \vdash \tau_1 : l_1 \qquad l_1 \vdash \tau_2 : l_2}{l \vdash \tau_1 \rightarrow \tau_2 : l_2} \tag{8}$$

$$\frac{l \vdash \tau_1 : l_1 \qquad l_1 \vdash \tau_2 : l_2}{l \vdash \tau_1 \times \tau_2 : l_2} \tag{9}$$

$$l \vdash decls[] : l \tag{10}$$

$$\frac{l \vdash \tau : l_1 \qquad l_1 \vdash \text{DECLS} : l_2}{l \vdash decls[\text{X} \mapsto \tau \cdot \text{DECLS}] : l_2} \tag{11}$$

end FREEVARS ;

Figure 4

set GEN is

$$\frac{\overset{\text{freevars}}{\vdash} \pi : l \qquad l \vdash \tau : \tau_1}{\pi \vdash \tau : \tau_1} \tag{1}$$

$$\frac{ISVAR(\tau :) \qquad ISIN(l, \tau :)}{l \vdash \tau : \tau} \tag{2}$$

$$\frac{ISVAR(\tau :)}{l \vdash \tau : gen(\tau)} \tag{3}$$

$$l \vdash int : int \tag{4}$$

$$l \vdash bool : bool \tag{5}$$

$$\frac{l \vdash \tau_1 : \tau_3 \qquad l \vdash \tau_2 : \tau_4}{l \vdash \tau_1 \rightarrow \tau_2 : \tau_3 \rightarrow \tau_4} \tag{6}$$

$$\frac{l \vdash \tau_1 : \tau_3 \qquad l \vdash \tau_2 : \tau_4}{l \vdash \tau_1 \times \tau_2 : \tau_3 \times \tau_4} \tag{7}$$

end GEN ;

set RENAME is

$$\frac{\tau \vdash decls[] : \tau_1, \pi_1}{\vdash \tau : \tau_1} \tag{1}$$

$$\frac{ISVAR(\tau :)}{\pi \vdash \tau : \tau, \pi} \tag{2}$$

$$\pi \vdash int : int, \pi \tag{3}$$

$$\pi \vdash bool : bool, \pi \tag{4}$$

$$\frac{\pi \vdash \tau_1 : \tau_3, \pi_1 \qquad \pi_1 \vdash \tau_2 : \tau_4, \pi_2}{\pi \vdash \tau_1 \rightarrow \tau_2 : \tau_3 \rightarrow \tau_4, \pi_2} \tag{5}$$

$$\frac{\pi \vdash \tau_1 : \tau_3, \pi_1 \qquad \pi_1 \vdash \tau_2 : \tau_4, \pi_2}{\pi \vdash \tau_1 \times \tau_2 : \tau_3 \times \tau_4, \pi_2} \tag{6}$$

$$\frac{\pi \overset{\text{nameof}}{\vdash} gen(\tau) : \tau_1}{\pi \vdash gen(\tau) : \tau_1, \pi} \tag{7}$$

$$\frac{\pi \overset{\text{newname}}{\vdash} gen(\tau), \tau_1 : \pi_1}{\pi \vdash gen(\tau) : \tau_1, \pi_1} \tag{8}$$

set NEWNAME is

$$\text{DECLS} \vdash \text{X}, \tau : decls[\text{X} \mapsto \tau \cdot \text{DECLS}] \tag{1}$$

end NEWNAME ;

Figure 4

set NAMEOF is

$$\frac{SAMEVAR(x_1, \mathrm{X} :)}{decls[\mathrm{X}_1 \mapsto \tau \cdot \mathrm{DECLS}] \vdash \mathrm{X} : \tau} \quad (1)$$

$$\frac{\mathrm{DECLS} \vdash \mathrm{X} : \tau}{decls[\mathrm{DECL} \cdot \mathrm{DECLS}] \vdash \mathrm{X} : \tau} \quad (2)$$

end NAMEOF ;

end RENAME ;

end L_TC

Figure 5

program ASPLE_SML is
use ASPLE **renaming** **program** **in** **asple_program** ;
use *SML* **renaming** *program* **in** *sml_program* ;
use **STORE**
$c, c_1, c_2, c_3 : SML;$
$s, s_1, s_2 : STORE;$

$$\frac{\mathbf{store}[\,] \vdash \mathrm{DECLS} : s \qquad \vdash \mathrm{STMS} : c}{\vdash \textit{begin}\ \mathrm{DECLS}\ \mathrm{STMS}\ \textit{end} : \textit{sml\_program}(c), s} \tag{1}$$

$$s \vdash \mathrm{decls}[\,] : s \tag{2}$$

$$\frac{s \vdash \mathrm{DECL} : s_1 \qquad s_1 \vdash \mathrm{DECLS} : s_2}{s \vdash \mathrm{DECL}; \mathrm{DECLS} : s_2} \tag{3}$$

$$\frac{s \vdash \mathrm{IDLIST} : s_1}{s \vdash \mathrm{MODE}\ \mathrm{IDLIST} : s_1} \tag{4}$$

$$\frac{s \overset{\text{allocate}}{\vdash} \mathbf{s\_id}\,\mathrm{ID} : s_1}{s \vdash \mathrm{id}\,\mathrm{ID} : s_1} \tag{5}$$

$$\frac{s \overset{\text{allocate}}{\vdash} \mathbf{s\_id}\,\mathrm{ID} : s_1 \qquad s_1 \vdash \mathrm{IDLIST} : s_2}{s \vdash \mathrm{id}\,\mathrm{ID}, \mathrm{IDLIST} : s_2} \tag{6}$$

$$\vdash \mathrm{stms}[\,] : \textit{coms}[\,] \tag{7}$$

$$\frac{\vdash \mathrm{STM} : c_1 \qquad \vdash \mathrm{STMS} : c_2 \qquad c = \textit{coms}[c_1; c_2]}{\vdash \mathrm{STM}; \mathrm{STMS} : c} \tag{8}$$

$$\frac{\vdash \mathrm{EXP} : c \qquad c_1 = \textit{coms}[c; \textit{sro}(\textit{sml\_id}\ \mathrm{ID})]}{\vdash \mathrm{id}\,\mathrm{ID} := \mathrm{EXP} : c_1} \tag{9}$$

$$\frac{\vdash \mathrm{EXP} : c_1 \qquad \vdash \mathrm{STMS} : c_2 \qquad c = \textit{coms}[c_1; \textit{fjp}\ 1; c_2; \textit{lbl}\ 1]}{\vdash \mathrm{if}\ \mathrm{EXP}\ \mathrm{then}\ \mathrm{STMS}\ \mathrm{fi} : \textit{block}(c)} \tag{10}$$

$$\frac{\vdash \mathrm{EXP} : c_1 \qquad \vdash \mathrm{STMS}_1 : c_2 \qquad \vdash \mathrm{STMS}_2 : c_3 \\ c = \textit{coms}[c_1; \textit{fjp}\ 1; c_2; \textit{ujp}\ 2; \textit{lbl}\ 1; c_3; \textit{lbl}\ 2]}{\vdash \mathrm{if}\ \mathrm{EXP}\ \mathrm{then}\ \mathrm{STMS}_1\ \mathrm{else}\ \mathrm{STMS}_2\ \mathrm{fi} : \textit{block}(c)} \tag{11}$$

$$\frac{\vdash \mathrm{EXP} : c_1 \qquad \vdash \mathrm{STMS} : c_2 \qquad c = \textit{coms}[\textit{lbl}\ 1; c_1; \textit{fjp}\ 2; c_2; \textit{ujp}\ 1; \textit{lbl}\ 2]}{\vdash \mathrm{while}\ \mathrm{EXP}\ \mathrm{do}\ \mathrm{STMS}\ \mathrm{end} : \textit{block}(c)} \tag{12}$$

$$\frac{\vdash \mathrm{EXP} : c \qquad c_1 = \textit{coms}[c; \textit{s\_read}]}{\vdash \mathrm{tinput}(\mathrm{EXP}, \mathrm{MODE}) : c_1} \tag{13}$$

$$\frac{\vdash \mathrm{EXP} : c \qquad c_1 = \textit{coms}[c; \textit{s\_write}]}{\vdash \mathrm{toutput}(\mathrm{EXP}, \mathrm{MODE}) : c_1} \tag{14}$$

$$\vdash \mathrm{id}\,\mathrm{X} : \textit{lao}(\textit{sml\_id}\ \mathrm{X}) \tag{15}$$

$$\frac{\vdash \mathrm{EXP} : c \qquad c_1 = \textit{coms}[c; \textit{ind}]}{\vdash \mathrm{deref}(\mathrm{EXP}) : c_1} \tag{16}$$

Figure 5

$$\vdash \textbf{boolean}\ \mathrm{V} : \mathit{ldci}(\mathit{sml\_boolean}\ \mathrm{V}) \qquad (17)$$

$$\vdash \textbf{number}\ \mathrm{N} : \mathit{ldci}(\mathit{sml\_number}\ \mathrm{N}) \qquad (18)$$

$$\frac{\vdash \mathrm{EXP}_1 : c_1 \qquad \vdash \mathrm{EXP}_2 : c_2 \qquad \overset{\text{operator}}{\vdash}\ \mathrm{OP} : \mathrm{OP}_1 \qquad c = \mathit{cons}[c_1; c_2; \mathrm{OP}_1]}{\vdash \mathrm{EXP}_1\ \mathrm{OP}\ \mathrm{EXP}_2 : c} \qquad (19)$$

set OPERATOR is

$$\vdash + : \mathit{op}\ \text{“adi”} \qquad (1)$$

$$\vdash \times : \mathit{op}\ \text{“mpi”} \qquad (2)$$

$$\vdash =: \mathit{op}\ \text{“equi”} \qquad (3)$$

$$\vdash \neq: \mathit{op}\ \text{“neqi”} \qquad (4)$$

$$\vdash \textbf{and} : \mathit{op}\ \text{“land”} \qquad (5)$$

$$\vdash \textbf{or} : \mathit{op}\ \text{“lor”} \qquad (6)$$

end OPERATOR ;

set ALLOCATE is

$$s \vdash \mathrm{S\_ID} : \mathbf{store}[\mathrm{S\_ID} \mapsto \mathbf{undef\_value} \cdot s] \qquad (1)$$

end ALLOCATE ;

end ASPLE_SML

Figure 6

program L_CAM is
use L
use *CAM*
$c, c_1, c_2, c_3 : CAM;$
$\rho, \rho_1, \rho_2 : ENV;$

$$\frac{\mathbf{nullpat} \vdash \mathrm{L\_exp} : c}{\mathrm{L\_exp} : program(c)} \tag{1}$$

$$\rho \vdash \mathbf{number}\ \mathrm{N} : coms[quote(\mathbf{int}\ \mathrm{N})] \tag{2}$$

$$\rho \vdash \mathbf{true} : coms[quote(bool\ \text{“true”})] \tag{3}$$

$$\rho \vdash \mathbf{false} : coms[quote(bool\ \text{“false”})] \tag{4}$$

$$\frac{\rho \overset{\mathrm{access}}{\vdash} \mathbf{ident}\ \mathrm{I} : c}{\rho \vdash \mathbf{ident}\ \mathrm{I} : c} \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 : c_1 \qquad \rho \vdash \mathrm{E}_2 : c_2 \qquad \rho \vdash \mathrm{E}_3 : c_3 \qquad c = coms[push; c_1; \mathbf{branch}(c_2, c_3)]}{\rho \vdash \mathbf{if}\ \mathrm{E}_1\ \mathbf{then}\ \mathrm{E}_2\ \mathbf{else}\ \mathrm{E}_3 : c} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 : c_1 \qquad \rho \vdash \mathrm{E}_2 : c_2 \qquad c = coms[push; c_1; swap; c_2; \mathbf{cons}]}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : c} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 : c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 : c_2 \qquad c = coms[push; c_1; cons; c_2]}{\rho \vdash \mathbf{let}\ \mathrm{P} = \mathrm{E}_1\ \mathbf{in}\ \mathrm{E}_2 : c} \tag{8}$$

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E}_1 : coms[cur(c_1)] \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 : c_2 \qquad c = coms[push; recf(c_1); cons; c_2]}{\rho \vdash \mathbf{letrec}\ \mathrm{P} = \mathrm{E}_1\ \mathbf{in}\ \mathrm{E}_2 : c} \tag{9}$$

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E} : c}{\rho \vdash \lambda \mathrm{P}.\mathrm{E} : coms[cur(c)]} \tag{10}$$

$$\frac{IS\_CONST(\mathrm{E}_1 :) \qquad \rho \vdash \mathrm{E}_2 : c_2 \qquad \overset{\mathrm{trans\_const}}{\vdash} \mathrm{E}_1 : c_1 \qquad c = coms[c_2; c_1]}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : c} \tag{11}$$

$$\frac{\rho \vdash \mathrm{E}_1 : c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 : c_2 \qquad c = coms[push; c_1; cons; c_2]}{\rho \vdash (\lambda \mathrm{P}.\mathrm{E}_2)\mathrm{E}_1 : c} \tag{12}$$

$$\frac{\rho \vdash \mathrm{E}_1 : c_1 \qquad \rho \vdash \mathrm{E}_2 : c_2 \qquad c = coms[push; c_1; swap; c_2; cons; \mathbf{app}]}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : c} \tag{13}$$

set ACCESS is

$$\mathbf{ident}\ \mathrm{X} \vdash \mathbf{ident}\ \mathrm{X} : coms[] \tag{1}$$

$$\frac{\rho_2 \vdash \mathrm{X} : c_2 \qquad c = coms[cdr; c_2]}{(\rho_1, \rho_2) \vdash \mathrm{X} : c} \tag{2}$$

$$\frac{\rho_1 \vdash \mathrm{X} : c_1 \qquad c = coms[car; c_1]}{(\rho_1, \rho_2) \vdash \mathrm{X} : c} \tag{3}$$

end ACCESS ;

Figure 6

set IS_CONST is

$\vdash$ ident "plus" (1)

$\vdash$ ident "minus" (2)

$\vdash$ ident "times" (3)

$\vdash$ ident "equal" (4)

$\vdash$ ident "fst" (5)

$\vdash$ ident "snd" (6)

end IS_CONST ;

set TRANS_CONST is

$\vdash$ ident "fst" : *car* (1)

$\vdash$ ident "snd" : *cdr* (2)

$\vdash$ ident X : *op* X (3)

end TRANS_CONST ;
end L_CAM

Figure 7

program SML_DS is
use SML
use *STORE*
$x, \varphi, \varphi_1, \varphi_2 : VALUE;$
$s, s_1, s_2 : STATE;$
$r, r_1, r_2 : STORE;$
$k : STACK;$
$\rho, \rho_1 : ENV;$
$c, c_1 : SML;$

$$\frac{env[\,] \vdash \text{COMS}, <r, [\,]> : s_1}{r \vdash \textbf{program}(\text{COMS})} \tag{1}$$

$$\rho \vdash \textbf{coms}[\,], s : s \tag{2}$$

$$\frac{\rho \overset{\texttt{cont\_find}}{\vdash} \textbf{lbl}\,\text{L} : c_1 \qquad \rho \vdash c_1, s : s_1}{\rho \vdash \textbf{coms}[\textbf{ujp}\,\text{L} \cdot c], s : s_1} \tag{3}$$

$$\frac{\rho \vdash c, <r, k> : s}{\rho \vdash \textbf{coms}[\textbf{fjp}\,\text{L} \cdot c], <r, \textit{boolean\_value}\text{ “true ”} \cdot k> : s} \tag{4}$$

$$\frac{\rho \overset{\texttt{cont\_find}}{\vdash} \textbf{lbl}\,\text{L} : c_1 \qquad \rho \vdash c_1, <r, k> : s}{\rho \vdash \textbf{coms}[\textbf{fjp}\,\text{L} \cdot c], <r, \textit{boolean\_value}\text{ “false ”} \cdot k> : s} \tag{5}$$

$$\frac{\rho \overset{\texttt{cont\_find}}{\vdash} \textbf{lbl}\,\text{L} : c_1 \qquad \rho \vdash c_1, <r, k> : s}{\rho \vdash \textbf{coms}[\textbf{tjp}\,\text{L} \cdot c], <r, \textit{boolean\_value}\text{ “true ”} \cdot k> : s} \tag{6}$$

$$\frac{\rho \vdash c, <r, k> : s}{\rho \vdash \textbf{coms}[\textbf{tjp}\,\text{L} \cdot c], <r, \textit{boolean\_value}\text{ “false ”} \cdot k> : s} \tag{7}$$

$$\frac{\rho \vdash \text{COM}, s : s_1 \qquad \rho \vdash \text{COMS}, s_1 : s_2}{\rho \vdash \textbf{coms}[\text{COM} \cdot \text{COMS}], s : s_2} \tag{8}$$

$$\frac{\overset{\texttt{cons\_env}}{\vdash} \text{COMS} : \rho_1 \qquad \rho_1 \vdash \text{COMS}, s : s_1}{\rho \vdash \textbf{block}(\text{COMS}), s : s_1} \tag{9}$$

$$\rho \vdash \textbf{nop}, s : s \tag{10}$$

$$\rho \vdash \textbf{ldci}(\textbf{sml\_number}\,\varphi), <r, k> : <r, \textit{number\_value}\,\varphi \cdot k> \tag{11}$$

$$\rho \vdash \textbf{ldci}(\textbf{sml\_boolean}\,\varphi), <r, k> : <r, \textit{boolean\_value}\,\varphi \cdot k> \tag{12}$$

$$\frac{r \overset{\texttt{get}}{\vdash} \textit{s\_id}\,\text{ID} : \varphi}{\rho \vdash \textbf{ldo}(\textbf{sml\_id}\,\text{ID}), <r, k> : <r, \varphi \cdot k>} \tag{13}$$

$$\frac{r \overset{\texttt{update}}{\vdash} \textit{s\_id}\,\text{ID}, \varphi : r_1}{\rho \vdash \textbf{sro}(\textbf{sml\_id}\,\text{ID}), <r, \varphi \cdot k> : <r_1, k>} \tag{14}$$

$$\rho \vdash \textbf{lao}(\textbf{sml\_id}\,\text{ID}), <r, k> : <r, \textit{s\_id}\,\text{ID} \cdot k> \tag{15}$$

Figure 7

$$\frac{r \overset{\text{get}}{\vdash} x : \varphi}{\rho \vdash \text{ind}, <r, x \cdot k> : <r, \varphi \cdot k>} \tag{16}$$

$$\frac{r \overset{\text{update}}{\vdash} x, \varphi : r_1}{\rho \vdash \text{sto}, <r, \varphi \cdot x \cdot k> : <r_1, k>} \tag{17}$$

$$\frac{\overset{\text{eval}}{\vdash} \varphi_1, \text{OP}, \varphi_2 : \varphi}{\rho \vdash \text{op}\,\text{OP}, <r, \varphi_1 \cdot \varphi_2 \cdot k> : <r, \varphi \cdot k>} \tag{18}$$

$$\rho \vdash \text{lbl}\,\text{L}, s : s \tag{19}$$

$$\frac{r \overset{\text{update}}{\vdash} x, \varphi : r_1 \qquad \overset{\text{s\_read}}{\vdash} : \varphi}{\rho \vdash \text{s\_read}, <r, x \cdot k> : <r_1, k>} \tag{20}$$

$$\frac{S\_WRITE(\varphi :)}{\rho \vdash \text{s\_write}, <r, \varphi \cdot k> : <r, k>} \tag{21}$$

set CONT_FIND is

$$env[\text{lbl}\,\text{L} \mapsto \text{CONT} \cdot \rho] \vdash \text{lbl}\,\text{L} : \text{CONT} \tag{1}$$

$$\frac{\rho \vdash \text{L} : \text{CONT}}{env[\text{PAIR} \cdot \rho] \vdash \text{L} : \text{CONT}} \tag{2}$$

end CONT_FIND ;

set CONS_ENV is

$$\vdash \text{coms}[\,] : env[\,] \tag{1}$$

$$\frac{\vdash \text{COMS} : \rho}{\vdash \text{coms}[\text{lbl}\,\text{L} \cdot \text{COMS}] : env[\text{lbl}\,\text{L} \mapsto \text{COMS} \cdot \rho]} \tag{2}$$

$$\frac{\vdash \text{COMS} : \rho}{\vdash \text{coms}[\text{COM} \cdot \text{COMS}] : \rho} \tag{3}$$

end CONS_ENV ;

set GET is

$$store[\text{X} \mapsto \varphi \cdot r] \vdash \text{X} : \varphi \tag{1}$$

$$\frac{r \vdash \text{X} : \varphi}{store[\text{VAL} \cdot r] \vdash \text{X} : \varphi} \tag{2}$$

end GET ;

Figure 7

set UPDATE is

$$\textit{store}[\mathrm{X} \mapsto \varphi_1 \cdot r] \vdash \mathrm{X}, \varphi_2 : \textit{store}[\mathrm{X} \mapsto \varphi_2 \cdot r] \tag{1}$$

$$\frac{r_1 \vdash \mathrm{X}, \varphi : r_2}{\textit{store}[\mathrm{VAL} \cdot r_1] \vdash \mathrm{X}, \varphi : \textit{store}[\mathrm{VAL} \cdot r_2]} \tag{2}$$

end UPDATE ;

end SML_DS

Figure 8

program CAM_DS is
use CAM
$s, s_1, s_2 : STACK;$
$\alpha, \beta, g, v : VALUE;$
$\rho, \rho_1, \rho_2 : ENV;$

$$\frac{\mathit{null\_value} \vdash \mathrm{COMS} : v \cdot \mathrm{s}}{\vdash \mathit{program}(\mathrm{COMS}) : v} \tag{1}$$

$$s \vdash \mathit{coms}[] : s \tag{2}$$

$$\frac{s \vdash \mathrm{COM} : s_1 \qquad s_1 \vdash \mathrm{COMS} : s_2}{s \vdash \mathit{coms}[\mathrm{COM} \cdot \mathrm{COMS}] : s_2} \tag{3}$$

$$\alpha \cdot \mathrm{s} \vdash \mathit{quote}(v) : v \cdot \mathrm{s} \tag{4}$$

$$<\alpha, \beta> \cdot\, \mathrm{s} \vdash \mathit{car} : \alpha \cdot \mathrm{s} \tag{5}$$

$$<\alpha, \beta> \cdot\, \mathrm{s} \vdash \mathit{cdr} : \beta \cdot \mathrm{s} \tag{6}$$

$$\alpha \cdot \beta \cdot \mathrm{s} \vdash \mathit{cons} : <\beta, \alpha> \cdot\, \mathrm{s} \tag{7}$$

$$\frac{\overset{\mathrm{eval}}{\vdash} \alpha, \mathrm{OP}, \beta : g}{<\alpha, \beta> \cdot\, \mathrm{s} \vdash \mathit{op}\ \mathrm{OP} : g \cdot \mathrm{s}} \tag{8}$$

$$\alpha \cdot \mathrm{s} \vdash \mathit{push} : \alpha \cdot \alpha \cdot \mathrm{s} \tag{9}$$

$$\alpha \cdot \beta \cdot \mathrm{s} \vdash \mathit{swap} : \beta \cdot \alpha \cdot \mathrm{s} \tag{10}$$

$$\frac{\rho_1 = \mathit{closure}(\mathrm{C}, \rho)}{\rho \cdot \mathrm{s} \vdash \mathit{cur}(\mathrm{C}) : \rho_1 \cdot \mathrm{s}} \tag{11}$$

$$\frac{\rho_1 = \mathit{closure}(\mathrm{C}, <\rho, \rho_1>)}{\rho \cdot \mathrm{s} \vdash \mathit{recf}(\mathrm{C}) : \rho_1 \cdot \mathrm{s}} \tag{12}$$

$$\frac{<\rho, \alpha> \cdot\, \mathrm{s} \vdash \mathrm{C} : s}{<\mathit{closure}(\mathrm{C}, \rho), \alpha> \cdot\, \mathrm{s} \vdash \mathbf{app} : s} \tag{13}$$

$$\frac{\mathrm{s} \vdash \mathrm{C}_1 : s_1}{\mathit{bool}\text{“true ”} \cdot \mathrm{s} \vdash \mathit{branch}(\mathrm{C}_1, \mathrm{C}_2) : s_1} \tag{14}$$

$$\frac{\mathrm{s} \vdash \mathrm{C}_2 : s_1}{\mathit{bool}\text{“false ”} \cdot \mathrm{s} \vdash \mathit{branch}(\mathrm{C}_1, \mathrm{C}_2) : s_1} \tag{15}$$

end CAM_DS

Figure 9

program CAM_DS is
use CAM
$s, s_1, s_2 : \mathit{STACK};$
$\alpha, \beta, g, v : \mathit{VALUE};$
$\rho, \rho_1, \rho_2 : \mathit{ENV};$

$$\frac{\mathit{null\_value} \vdash \textsc{coms} : v \cdot \mathrm{s}}{\vdash \mathit{program}(\textsc{coms}) : v} \tag{1}$$

$$s \vdash \mathit{coms}[] : s \tag{2}$$

$$\frac{s \vdash \textsc{com} : s_1 \qquad s_1 \vdash \textsc{coms} : s_2}{s \vdash \mathit{coms}[\textsc{com} \cdot \textsc{coms}] : s_2} \tag{3}$$

$$\alpha \cdot \mathrm{s} \vdash \mathit{quote}(v) : v \cdot \mathrm{s} \tag{4}$$

$$<\alpha, \beta> \cdot\, \mathrm{s} \vdash \mathit{car} : \alpha \cdot \mathrm{s} \tag{5}$$

$$<\alpha, \beta> \cdot\, \mathrm{s} \vdash \mathit{cdr} : \beta \cdot \mathrm{s} \tag{6}$$

$$\alpha \cdot \beta \cdot \mathrm{s} \vdash \mathit{cons} : <\beta, \alpha> \cdot\, \mathrm{s} \tag{7}$$

$$\frac{\overset{\text{eval}}{\vdash} \alpha, \textsc{op}, \beta : g}{<\alpha, \beta> \cdot\, \mathrm{s} \vdash \mathit{op}\ \textsc{op} : g \cdot \mathrm{s}} \tag{8}$$

$$\alpha \cdot \mathrm{s} \vdash \mathit{push} : \alpha \cdot \alpha \cdot \mathrm{s} \tag{9}$$

$$\alpha \cdot \beta \cdot \mathrm{s} \vdash \mathit{swap} : \beta \cdot \alpha \cdot \mathrm{s} \tag{10}$$

$$\rho \cdot \mathrm{s} \vdash \mathit{cur}(\mathrm{C}) : \mathit{closure}(\mathrm{C}, \rho) \cdot \mathrm{s} \tag{11}$$

$$\rho \cdot \mathrm{s} \vdash \mathit{recf}(\mathrm{C}) : \mathit{recclosure}(\mathrm{C}, \rho) \cdot \mathrm{s} \tag{12}$$

$$\frac{<\rho, \alpha> \cdot\, \mathrm{s} \vdash \mathrm{C} : s}{<\mathit{closure}(\mathrm{C}, \rho), \alpha> \cdot\, \mathrm{s} \vdash \mathbf{app} : s} \tag{13}$$

$$\frac{<<\rho, \mathit{recclosure}(\mathrm{C}, \rho)>, \alpha> \cdot\, \mathrm{s} \vdash \mathrm{C} : s}{<\mathit{recclosure}(\mathrm{C}, \rho), \alpha> \cdot\, \mathrm{s} \vdash \mathbf{app} : s} \tag{14}$$

$$\frac{\mathrm{s} \vdash \mathrm{C}_1 : s_1}{\mathit{bool}\ \text{“true ”} \cdot \mathrm{s} \vdash \mathbf{branch}(\mathrm{C}_1, \mathrm{C}_2) : s_1} \tag{15}$$

$$\frac{\mathrm{s} \vdash \mathrm{C}_2 : s_1}{\mathit{bool}\ \text{“false ”} \cdot \mathrm{s} \vdash \mathbf{branch}(\mathrm{C}_1, \mathrm{C}_2) : s_1} \tag{16}$$

end CAM_DS

Figure 10

program ASPLE_DS is
use ASPLE
use *STORE*
x, x_1, x_2, v, v_1 : *VALUE*;
s, s_1, s_2 : *STORE*;

$$\frac{\mathit{store}[] \vdash \text{DECLS} : s_1 \qquad s_1 \vdash \text{STMS} : s_2}{\vdash \text{begin DECLS STMS end}} \tag{1}$$

$$s \vdash \text{decls}[] : s \tag{2}$$

$$\frac{s \vdash \text{DECL} : s_1 \qquad s_1 \vdash \text{DECLS} : s_2}{s \vdash \text{DECL};\text{DECLS} : s_2} \tag{3}$$

$$\frac{s \vdash \text{IDLIST} : s_1}{s \vdash \text{MODE IDLIST} : s_1} \tag{4}$$

$$\frac{s \overset{\text{allocate}}{\vdash} s\_id\ \text{ID} : s_1}{s \vdash \text{id ID} : s_1} \tag{5}$$

$$\frac{s \overset{\text{allocate}}{\vdash} s\_id\ \text{ID} : s_1 \qquad s_1 \vdash \text{IDLIST} : s_2}{s \vdash \text{id ID}, \text{IDLIST} : s_2} \tag{6}$$

$$s \vdash \text{stms}[] : s \tag{7}$$

$$\frac{s \vdash \text{STM} : s_1 \qquad s_1 \vdash \text{STMS} : s_2}{s \vdash \text{STM};\text{STMS} : s_2} \tag{8}$$

$$\frac{s \vdash \text{EXP} : v \qquad s \overset{\text{update}}{\vdash} s\_id\ \text{ID}, v : s_1}{s \vdash \text{id ID} := \text{EXP} : s_1} \tag{9}$$

$$\frac{s \vdash \text{EXP} : \mathit{boolean\_value}\ \text{“true”} \qquad s \vdash \text{STMS}_1 : s_1}{s \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1} \tag{10}$$

$$\frac{s \vdash \text{EXP} : \mathit{boolean\_value}\ \text{“false”} \qquad s \vdash \text{STMS}_2 : s_1}{s \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1} \tag{11}$$

$$\frac{s \vdash \text{EXP} : \mathit{boolean\_value}\ \text{“true”} \qquad s \vdash \text{STMS} : s_1}{s \vdash \text{if EXP then STMS fi} : s_1} \tag{12}$$

$$\frac{s \vdash \text{EXP} : \mathit{boolean\_value}\ \text{“false”}}{s \vdash \text{if EXP then STMS fi} : s} \tag{13}$$

$$\frac{s \vdash \text{EXP} : \mathit{boolean\_value}\ \text{“true”} \qquad s \vdash \text{STMS} : s_1 \qquad s_1 \vdash \text{while EXP do STMS end} : s_2}{s \vdash \text{while EXP do STMS end} : s_2} \tag{14}$$

$$\frac{s \vdash \text{EXP} : \mathit{boolean\_value}\ \text{“false”}}{s \vdash \text{while EXP do STMS end} : s} \tag{15}$$

$$\frac{s \vdash \text{EXP} : s\_id\ \text{X} \qquad s \overset{\text{update}}{\vdash} s\_id\ \text{X}, v : s_1 \qquad \overset{\text{a\_read}}{\vdash} \text{MODE} : v}{s \vdash \text{tinput}(\text{EXP}, \text{MODE}) : s_1} \tag{16}$$

Figure 10

$$\frac{s \vdash \text{EXP} : v \qquad A\_WRITE(v :)}{s \vdash \text{toutput}(\text{EXP}, \text{MODE}) : s} \tag{17}$$

$$s \vdash \text{id}\, \text{X} : s\_id\, \text{X} \tag{18}$$

$$\frac{s \vdash \text{EXP} : s\_id\, \text{X} \qquad s \overset{\text{store}}{\vdash} s\_id\, \text{X} : v}{s \vdash \text{deref}(\text{EXP}) : v} \tag{19}$$

$$s \vdash \text{boolean}\, \text{X} : boolean\_value\, \text{X} \tag{20}$$

$$s \vdash \text{number}\, \text{X} : number\_value\, \text{X} \tag{21}$$

$$\frac{s \vdash \text{EXP}_1 : x_1 \qquad s \vdash \text{EXP}_2 : x_2 \qquad \overset{\text{eval}}{\vdash} x_1, \text{OP}, x_2 : x}{s \vdash \text{EXP}_1\, \text{OP}\, \text{EXP}_2 : x} \tag{22}$$

set ALLOCATE is

$$s \vdash \text{S\_ID} : store[\text{S\_ID} \mapsto undef\_value \cdot s] \tag{1}$$

end ALLOCATE ;

set UPDATE is

$$store[\text{S\_ID} \mapsto \text{V} \cdot \text{S}] \vdash \text{S\_ID}, \text{V}_1 : store[\text{S\_ID} \mapsto \text{V}_1 \cdot \text{S}] \tag{1}$$

$$\frac{\text{S} \vdash \text{S\_ID}, v_1 : \text{S}_1}{store[\text{V} \cdot \text{S}] \vdash \text{S\_ID}, v_1 : store[\text{V} \cdot \text{S}_1]} \tag{2}$$

end UPDATE ;

set STORE is

$$store[\text{S\_ID} \mapsto \text{V} \cdot \text{S}] \vdash \text{S\_ID} : \text{V} \tag{1}$$

$$\frac{\text{S} \vdash \text{S\_ID} : v_1}{store[\text{V} \cdot \text{S}] \vdash \text{S\_ID} : v_1} \tag{2}$$

end STORE ;

end ASPLE_DS

Figure 11

program L_DS is
use L
$\rho, \rho_1, \rho_2 : \mathit{ENV};$
$\alpha, \beta, \gamma : \mathit{VALUE};$

$$\frac{\rho = \mathit{init\_env}() \qquad \rho \vdash \mathrm{L\_EXP} : \alpha}{\vdash \mathrm{L\_EXP} : \alpha} \tag{1}$$

$$\rho \vdash \mathbf{number}\ \mathrm{N} : \mathit{int}\ \mathrm{N} \tag{2}$$

$$\rho \vdash \mathbf{true} : \mathit{bool}\ \text{“true”} \tag{3}$$

$$\rho \vdash \mathbf{false} : \mathit{bool}\ \text{“false”} \tag{4}$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \mathbf{ident}\ \mathrm{I} : \alpha}{\rho \vdash \mathbf{ident}\ \mathrm{I} : \alpha} \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{bool}\ \text{“true”} \qquad \rho \vdash \mathrm{E}_2 : \alpha}{\rho \vdash \mathbf{if}\ \mathrm{E}_1\ \mathbf{then}\ \mathrm{E}_2\ \mathbf{else}\ \mathrm{E}_3 : \alpha} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{bool}\ \text{“false”} \qquad \rho \vdash \mathrm{E}_3 : \alpha}{\rho \vdash \mathbf{if}\ \mathrm{E}_1\ \mathbf{then}\ \mathrm{E}_2\ \mathbf{else}\ \mathrm{E}_3 : \alpha} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad \rho \vdash \mathrm{E}_2 : \beta}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad \mathit{env}[\mathrm{P} \mapsto \alpha \cdot \rho] \vdash \mathrm{E}_2 : \beta}{\rho \vdash \mathbf{let}\ \mathrm{P} = \mathrm{E}_1\ \mathbf{in}\ \mathrm{E}_2 : \beta} \tag{9}$$

$$\frac{\rho_1 = \mathit{env}[\mathrm{P} \mapsto \mathit{closure}(\mathrm{E}_1, \rho_1) \cdot \rho] \qquad \rho_1 \vdash \mathrm{E}_2 : \beta}{\rho \vdash \mathbf{letrec}\ \mathrm{P} = \mathrm{E}_1\ \mathbf{in}\ \mathrm{E}_2 : \beta} \tag{10}$$

$$\rho \vdash \lambda \mathrm{P.E} : \mathit{closure}(\lambda \mathrm{P.E}, \rho) \tag{11}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathbf{ident}\ \mathrm{OP} \qquad \rho \vdash \mathrm{E}_2 : (\alpha, \beta) \qquad \overset{\text{eval}}{\vdash} \alpha, \mathrm{OP}, \beta : \gamma}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \gamma} \tag{12}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad \mathit{env}[\mathrm{P} \mapsto \alpha \cdot \rho] \vdash \mathrm{E}_2 : \beta}{\rho \vdash (\lambda \mathrm{P.E}_2)\mathrm{E}_1 : \beta} \tag{13}$$

$$\frac{\rho \vdash \mathrm{E}_2 : \alpha \qquad \rho \vdash \mathrm{E}_1 : \mathit{closure}(\lambda \mathrm{P.E}, \rho_1) \qquad \rho_2 = \mathit{env}[\mathrm{P} \mapsto \alpha; \rho_1; \rho] \qquad \rho_2 \vdash \mathrm{E} : \beta}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \beta} \tag{14}$$

set VAL_OF is

$$\mathit{env}[\mathbf{ident}\ \mathrm{I} \mapsto \alpha \cdot \rho] \vdash \mathbf{ident}\ \mathrm{I} : \alpha \tag{1}$$

$$\frac{\mathit{env}[\mathrm{P}_1 \mapsto \alpha, \mathrm{P}_2 \mapsto \beta] \vdash \mathbf{ident}\ \mathrm{I} : \gamma}{\mathit{env}[(\mathrm{P}_1, \mathrm{P}_2) \mapsto (\alpha, \beta) \cdot \rho] \vdash \mathbf{ident}\ \mathrm{I} : \gamma} \tag{3}$$

Figure 11

$$\frac{\mathit{env}[\mathrm{P}_1 \mapsto \mathit{closure}(\mathrm{E}_1, \rho_1), \mathrm{P}_2 \mapsto \mathit{closure}(\mathrm{E}_2, \rho_1)] \vdash \mathbf{ident}\,\mathrm{I} : \gamma}{\mathit{env}[(\mathrm{P}_1, \mathrm{P}_2) \mapsto \mathit{closure}((\mathrm{E}_1, \mathrm{E}_2), \rho_1) \cdot \rho] \vdash \mathbf{ident}\,\mathrm{I} : \gamma} \tag{3}$$

$$\frac{\rho \vdash \mathbf{ident}\,\mathrm{I} : \alpha}{\mathit{env}[\mathrm{P} \cdot \rho] \vdash \mathbf{ident}\,\mathrm{I} : \alpha} \tag{4}$$

end VAL_OF ;
end L_DS

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique