STATIC TYPE CHECKING FOR LANGUAGES WITH PARAMETRIC TYPES AND POLYMORPHIC PROCEDURES

R. Barbuti (1) and A. Martelli (1)

Abstract : The paper presents a static type checking algorithm for a language which allows to declare, and to pass as parameters, types, type constructors (parametric types) and polimorphic functions. A program written in this language is translated into an expression in a suitable formalism, and this expression is reduced to a normal form which can be either error or a type correct expression. This approach can also be described as that of giving a non-standard interpretation to programs (in this case a symbolic interpretation) using a denotational semantics technique.

---

(1) : Istituto di Elaborazione della Informazione del CNR
Via S. Maria, 46
56100 PISA, Italy

1. INTRODUCTION

Types play an increasingly important role in programming languages. It is now widely accepted that a powerful type structure is an abstraction mechanism which adds expressive power to a language, thus making the writing of correct programs easier.

Most recently designed languages (such as Euclid (Lampson et al, 1977), Mesa (Mitchell et al, 1979), ADA (Ichbiah et al, 1979)) provide constructs for defining new data types together with their operations. Furthermore, some of these languages allow the definition of parametric types (such as array(n,m,t) where n and m can be any integer, and t any type), and polymorphic procedures, i.e. procedures where the type of some parameters is not completely specified (such as push(s,el), where s has type stack(t) and el has type t, for any t).

In languages with a type structure of this kind, strong type checking can catch a large number of programming errors. For this reason it is very important to be able to perform type checking at compile time, even if this entails some restrictions on the type structure. A further advantage of static type checking is of course computational efficiency, since run time checks are no longer necessary.

Static type checking is described in the literature for Algol 60 (Ledgard, 1972), Pascal (Ermine and Ressouche, 1978) and Algol 68-like languages (Henderson, 1977). In these papers, type checking is carried out by translating a program into an expression in a suitable formalism which characterizes the type structure, while disregarding all other irrelevant aspects. This technique might also be considered as a way of assigning a non-standard semantics (static semantics) to the program using a denotational approach, as pointed out, for instance, in (Donzeau-Gouge, 1978).

An algorithm for static type checking in a language with polymorphic procedures was proposed by Milner in (Milner, 1978). In this language the programmer never writes the type of a variable, and the type checker is able to prove the correctness of a program knowing the types of all primitive operations and data, making use of a unification algorithm.

In this paper we use an approach similar to Ledgard's (Ledgard, 1972) to perform static type checking on a language which allows the definition of

parametric types and polymorphic procedures. Differently from Milner's language, in this language the type of every identifier must be explicitly expressed, and types can be passed as parameters to polymorphic procedures, in order to specify the type of other parameters (as suggested in (Tennent, 1977), (Demers et al, 1978), (Asirelli et al, 1977)).

This language is described in the next section. It is a simple applicative language since in this paper we focus only on aspects related with types. However the same approach can easily be applied to an imperative language with a richer syntax. Section 3 describes the formalism into which the programs will be translated, and Section 4 contains the translation rules.

2. A SIMPLE LANGUAGE WITH PARAMETRIC TYPES AND POLYMORPHIC FUNCTIONS

In this section we give an informal description of a simple applicative language, which allows to define new data types, type contructors and polymorphic procedures. The type structure of this language has been extracted from the language described in (Asirelli et al, 1978) and (Asirelli et al, 1979), which was designed with the purpose of controlling the most common sources of side effects, and of allowing the definition of abstract data types.

Every denotable value has a type, and types are also denotable values (with type type). So for instance we can declare

let t : type = integer,

let x : t = 5.

Types can either be primitive types or they can be obtained by applying type constructors (parametric types) to their arguments. These arguments need not necessarily be types. Type constructors can be defined, in this language, exactly like functions. For instance

lambda (t:type,n:integer).pair(array(1,n,t),integer)

is a type constructor with two parameters, and thus by applying it, for instance, to the arguments (boolean,5) we get the type

pair(array(1,5,boolean),integer).

"pair" and "array" are previously declared (or primitive) type constructors.

Primitive types are all different, and the same type constructor applied to different parameters yields different types.

Since we have parametric types, we want also to be able to define polymorphic functions, i.e. functions whose parameter types are not completely specified. For instance the function which selects an element of an array accepts as actual parameters an integer value and an "array(n,m,t)" where n and m can be any integer, and t any type. To deal with this case we add n, m and t as further parameters (Asirelli et al, 1977), (Demers et al, 1978), (Reynolds, 1974), (Tennent, 1977):

lambda(n:integer,m:integer,t:type,a:array(n,m,t),index:integer) ...

In a function call, parameters are bound in the given order, and thus, when the fourth parameter is bound, its type is completely specified.

Types of functions and type constructors can also be expressed in the language. To allow a static type checking, the type of a function (or of a type constructor) must contain the types of all parameters (as for instance in Algol 68). Furthermore, as the above example shows, to express the type of a parameter we need the names of some other parameters, and thus the type of a function must contain also the names of the parameters. For instance, the type of the above function is

func(n:integer,m:integer,t:type,a:array(n,m,t),i:integer) → t

where → t is the type of the result. Similarly, the type of the type constructor given in the second example is

tconstr(t:type,n:integer).

In general a declaration has the form

let x : $expr_1$ = $expr_2$

where x is an identifier, and $expr_1$ and $expr_2$ are expressions. The value of $expr_2$ is bound to x if its type is equal to the value of $expr_1$; otherwise a type error occurs.

Parameter passing is performed in the same way. Actual parameters are evaluated and bound to formal parameters, if type checking succeeds. Every denotable value can also be passed as a parameter, in particular polymorphic functions and type constructors. For instance we can define the following type constructor

```
let double:tconstr(c:tconstr(t:type),t1:type)=
    lambda(c:tconstr(t:type),t1:type).c(c(t1))
```

which applies twice its parameter c to the other parameter t1.

Recursive functions can be declared using the letrec construct, and no other value can be declared recursively. In particular recursive definitions of types or type constructors are not allowed. In fact the presence of recursively defined types would yield very hard equivalence problems (Solomon, 1978).

The type (or type constructor) declarations previously shown do not create new types (or type constructors), but they simply give a name to a type value. For instance, in

```
let t : type = integer,
let x : t = 5
```

the second declaration is type correct, because t is bound to "integer".

We can also declare new types (or type constructors) by the new construct. For instance, the declaration

```
new abst : type = integer
```

creates a new (abstract) type, whose (concrete) representation is "integer". Now the declaration

```
let x : abst = 5
```

would be wrong.

To access the representation of an abstract type (or type constructor) t we use the notation ↓t. For instance

```
let x : ↓abst = 5
```

is correct (since ↓abst is "integer").

Usually the parameters of functions which implement operations on abstract data types, have abstract type outside, and concrete type inside. Thus two functions "up" and "down" are provided to change types from concrete to abstract and vice versa. For instance if we have

```
new stack : tconstr(t:type) =
            lambda(t:type).pair(array(1,100,t),integer)
let y : stack(abst) = ...
```

then

```
down(y, ↓stack( ↓abst))
```

returns a value whose type is "pair(array(1,100,integer),integer)",whereas

```
down(y, ↓stack(abst))
```

returns a value with type "pair(array(1,100,abst),integer)".

As the above example shows, the function "down" goes down from the ab-

stract type to the representation for all types or type constructors whose name is prefixed with ↓.

The new construct for declaring abstract data types is not very useful in this language, because there is no way of hiding the representation. The language described in (Asirelli et al, 1979), as well as all languages mentioned in the introduction, provides a construct, the module, for defining abstract data types. Modules are not considered in this paper because they have to do only with scope rules and not with type checking.

New types and type constructors can be defined recursively. In fact the ↓ operator goes down in the representation only one level, and thus types remain always finite.

2.1 The syntax of the language

We give now the sintax of the language. The syntactic categories are

X ∈ IDE (identifiers)

E ∈ EXPR (expressions)

D ∈ DCL (declarations)

The sintax is

E::= X

type

(D_0,D_1,...,D_n) in E

lambda(X_0:E_0,X_1:E_1, ...,X_n:E_n).E

func(X_0:E_0,X_1:E_1, ...,X_n:E_n) → E

tconstr(X_0:E_0,X_1:E_1, ...,X_n:E_n)

E(E_0,E_1, ...,E_n)

if E_0 then E_1 else E_2

↓X

up(E_1,E_2)

down(E_1,E_2)

D::= let X:E_1=E_2

letrec X:func(X_0:E_0,X_1:E_1, ...,X_n:E_n) → E_{n+1}=E

new X:type=E

new X:tconstr(X_0:E_0,X_1:E_1, ...,X_n:E_n)=E

In order to perform type checking, an expression in this language is translated into an expression in a suitable formalism, a typed λ-calculus, which characterizes the type structure of the language while disregarding all other aspects.

3. A FORMALISM FOR TYPE CHECKING

This formalism is an extension of typed λ-calculus, where types are expressions of the formalism as well (similar extensions have been proposed in (Reynolds, 1974) and (Donahue, 1979)). The syntactic categories are

$x \in$ Ide (Identifiers)

$c \in$ Const (Constants)

$e \in$ Expr (Expressions)

The syntax of expressions is

e::= x

type

<c,e>

$\lambda x{:}e_1.e_2$

$\Lambda x{:}e_1.e_2$

$e_1(e_2)$

$e_1[e_2]$

error

As we will show in the next section, expressions <c,e> will denote typed constants, $\lambda x{:}e_1.e_2$ will denote function and type constructor values, whereas $\Lambda x{:}e_1.e_2$ will denote their type. Furthermore, $e_1(e_2)$ is the application of a λ-expression, whereas $e_1[e_2]$ is the application of a Λ-expression.

In order to perform type checking on expressions in this formalism, we define a function NF which reduces expressions to normal form. Function NF performs every possible (typed) β-reduction on the original expression, and furthermore propagates type errors, in such a way that type incorrect expressions will be reduced to error.

The type of an identifier occurring free in an expression is not known in the expression itself. Therefore function NF needs also a type environment which associates types with free identifiers. The two operations on a type en-

vironment "te" are a write operation (syntactically denoted by te{e/x}), and a read operation te(x), which returns error if x is not bound in te.

Function NF is defined as follows

$$NF(x)te = x$$

$$NF(\underline{type})te = \underline{type}$$

$$NF(\langle c,e\rangle)te = \underline{if}\ NF(e)te = \underline{error}\ \underline{then}\ \underline{error}\ \underline{else}\ \langle c, NF(e)te\rangle$$

$$NF(\lambda x{:}e_1.e_2)te = \underline{if}\ v_1{=}\underline{error}\ \text{or}\ v_2{=}\underline{error}\ \underline{then}\ \underline{error}\ \underline{else}\ \lambda x{:}v_1.v_2$$
$$\text{where } v_1{=}NF(e_1)te,\ v_2{=}NF(e_2)te\{v_1/x\}$$

$$NF(\Lambda x{:}e_1.e_2)te = \underline{if}\ v_1{=}\underline{error}\ \text{or}\ v_2{=}\underline{error}\ \underline{then}\ \underline{error}\ \underline{else}\ \Lambda x{:}v_1.v_2$$
$$\text{where } v_1{=}NF(e_1)te,\ v_2{=}NF(e_2)te\{v_1/x\}$$

$$NF(e_1(e_2))te = \underline{case}\ v_1\ \underline{of}$$
$$\underline{error} \rightarrow \underline{error};$$
$$\lambda x{:}e'.e'' \rightarrow \underline{if}\ e' \cong NF(typeof(v_2)te)te\ \underline{then}\ NF(\{v_2/x\}e'')te\ \underline{else}\ \underline{error};$$
$$\underline{else} \rightarrow \underline{if}\ NF((typeof(v_1)te)[v_2])te{=}\underline{error}\ \underline{then}\ \underline{error}\ \underline{else}\ v_1(v_2)$$
$$\text{where } v_1{=}NF(e_1)te,\ v_2{=}NF(e_2)\ te$$

$$NF(e_1[e_2])te = \underline{case}\ v_1\ \underline{of}$$
$$\Lambda x{:}e'.e'' \rightarrow \underline{if}\ e' \cong NF(typeof(v_2)te)te\ \underline{then}\ NF(\{v_2/x\}e'')te\ \underline{else}\ \underline{error}$$
$$\underline{else} \rightarrow \underline{error}$$
$$\text{where } v_1{=}NF(e_1)te,\ v_2{=}NF(e_2)te$$

$$NF(\underline{error})te = \underline{error}$$

Function "typeof" returns the type of any expression in normal form, and is defined as follows

typeof(x)te = te(x)

typeof(<u>type</u>)te = <u>error</u>

typeof(<c,e>)te = e

typeof(λx:e_1.e_2)te = Λx:e_1.typeof(e_2)te{e_1/x}

typeof(Λx:e_1.e_2)te = typeof(e_2)te{e_1/x}

typeof(e_1(e_2))te = (typeof(e_1)te) [e_2]

typeof(<u>error</u>)te = <u>error</u>

Note that typeof (e_1[e_2]) has not been defined because, according to the definition of NF, this application cannot occur in any expression in normal form.

Reduction to normal form of an application $e_1(e_2)$ (or $e_1[e_2]$), where e_1 is a λ (or Λ)-expression, is similar to β-reduction of λ-calculus, and thus the notation {v_2/x}e" means substituting v_2 for every occurrence of x in e". This reduction is correct if the types of the actual and the formal parameter are equal. Two type expressions t_1 and t_2 are equal ($t_1 \cong t_2$) if, by suitably renaming bound identifiers, they become identical.

When e_1 is not a λ-expression, an application $e_1(e_2)$ is correct if the type of e_1 can be correctly applied to e_2 (after reduction to normal form of all expressions).

To deal with the case of functions with more than one parameter, we extend our notation by allowing λ(or Λ)-expressions of the form

$\lambda(x_0:e_0, x_1:e_1, \ldots, x_n:e_n).e$

and applications of the form

$e(e_0, e_1, \ldots, e_n)$ or $e[e_0, e_1, \ldots, e_n]$

The meaning of the above expressions is respectively

$\lambda x_0:e_0.\ \lambda x_1:e_1.\ \ldots \lambda x_n:e_n.e$

and

$e(e_0)(e_1)\ldots(e_n)$ or $e[e_0][e_1]\ \ldots[e_n]$.

However an application is correct only if the list of actual and formal parameters have the same length.

4. TRANSLATION RULES

In this section we give the definition of a function TE which translates expressions of the language of Section 2 into expressions of the formalism described in the previous section. As usual in denotational semantics, the function TE uses an environment which, in our case, maps identifiers of the language into expressions in the formalism:

env ∈ U = [IDE → Expr]

Since function TE, besides translation, makes some checks on the resulting expressions, it needs the types of all free identifiers in these expressions. Therefore this function uses also a type environment te, as defined in the previous section:

te ∈ V = [Ide → Expr]

So the type of function TE is

TE : (EXPR x U x V) → Expr

Furthermore we need a translation function TD for declarations, which modifies the environment:

TD : (DCL x U x V) → U.

The definitions of the two functions TE and TD are given in Table 1. Here we make a few comments on these definitions.

The translation of all syntactic constructs containing a list of parameters (i.e. lambda, func, tconstr) makes use of a function "istype" (not defined here) which checks whether an expression is correctly used as a type, i.e. whether it is either type or a Λ-expression or its type is type. This function is also used in the translation of a let declaration.

The expression "if E_0 then E_1 else E_2" is translated into the application of a polymorphic function "cond" to the results of the translations of E_0, E_1, E_2. Function "cond" checks that E_0 has type "boolean" and that E_1 and E_2 have the same type. The result has also this type.

A recursive declaration "letrec X:E_1.E_2", where E_1 is a function type, is processed by assuming inductively that the type of X in every recursive call is E_1, and then checking that the type of E_2 is actually E_1. Constant "a" in the expression $\langle a,v_1 \rangle$ must be a new constant never used before. If type checking succeeds, X is bound to the expression resulting from the translation of E_2,

which is a finite expression approximating the infinite expression which should be the result of the recursive definition.

Declarations of new types (and similarly of new type constructors) are processed by inserting in the environment two new entries: the first entry associates with the name of the abstract type being defined a new constant with type type: the second entry associates with a new identifier, obtained by prefixing the name of the type with ↓, the representation of the type. Note that recursive definitions are allowed.

Finally, the definition of TE for "up" or "down" makes use of two different translations of the second parameter. In fact, whereas v_2 is the result of the usual translation of E_2, v'_2 is obtained by applying TE to a new expression obtained by removing all ↓ operators from E_2. By interpreting E_2 as a type expression, v_2 and v'_2 are the concrete and the abstract value, respectively.

5. CONCLUSIONS

In this paper we have presented an algorithm for static type checking which translates a program into an expression in a suitable formalism, and then reduces this expression either to error or to a type correct expression. This translation defines a non-standard semantics for the language (Donzeau-Gouge, 1978), by associating with programs values in the domain of symbolic expressions. Type checking is carried out by checking symbolic expressions for equality, and thus it has a fail-safe character: in fact, two expressions like stack(integer,n) and stack(integer,n+1-1) are not recognized to denote the same type. Furthermore in our type structure every value has exactly one type, and so it is not possible to define union of types or subtypes like Pascal subranges. If this were allowed, a static type checking would become much more complicated, or even impossible without the insertion of assertions in the program (Cousot and Cousot, 1977).

The type checking algorithm has been implemented for an extension of the language of this paper (Barbuti, 1979). The language has been extended in two directions. First, parameters can be passed to polymorphic procedures by pattern matching to avoid proliferation of parameters. Thus, instead of declaring

```
select:... =
 lambda(n:integer,m:integer,t:type,a:array(n,m,t),i:t)...
```

we allow also

```
select:... =
 lambda(a:array(<n:integer>,<m:integer>,<t:type>),i:t)...
```

Now "select" will be called with two, instead of five, actual parameters, and n, m and t will be passed by pattern matching. The formalism of Section 3 has been extended to deal with this situation.

The second extension regards the introduction of the store. This extension can be dealt with by the formalism of Section 3 without any modification, by adding, as usual in denotational semantics, the store as an additional parameter to all functions or type constructors.

REFERENCES

Asirelli P., Gimona F., Martelli A., Montanari U. Passing parameter types in programming languages with data abstractions. AICA '77 Congress, Pisa, October 1977, 429-444.

Asirelli P., Martelli A., Montanari U. Language constructs for controlling side effects. Internal Report OL78-6, IEI, Pisa, December 1978, Submitted for publication.

Asirelli P., Degano P., Levi G., Martelli A., Montanari U., Pacini G., Sirovich F., Turini F. A flexible environment for program development based on a symbolic interpreter. Proc. Fourth Int. Conference on Software Engineering, Munich, September 1979, 251-263.

Barbuti R. Verifica dei tipi statica per un linguaggio con procedure polimorfe. AICA '79 Congress, Bari, October 1979.

Cousot P., Cousot R. Static determination of dynamic properties of generalized type unions. Language Design for Reliable Software, SIGPLAN Notices, 12, 3, March 1977, 77-94.

Demers A., Donahue J., Skinner G. Data types as values: Polimorphism, Type-checking, Encapsulation. Proc. Fifth ACM Symposium on Principles of Programming Languages, Tucson, January 1978, 23-30.

Donahue J. On the semantics of "Data types". SIAM J. Computing, 8, 4, November 1979, 546-560.

Donzeau-Gouge V. Utilization de la semantique denotationelle pour la description d'interpretations non-standard : application a la validation et a l'optimisation des programmes. Third International Symp. on Programming, Paris, March 1978, 315-335.

Ermine F., Ressouche A. Une methode de verification statique de types. Application au langage Pascal. Third International Symp. on Programming, Paris, March 1978, 292-314.

Henderson P. An approach to compile time type checking. Information Processing 77, North Holland 1977, 523-527.

Ichbiah J.D. et al. Rationale for the design of the ADA programming language. SIGPLAN Notices, 14, 7, July 1979.

Lampson B.W., Horning J.J., London R.L., Mitchell J.G., Popek G.L. Report on the programming language Euclid. SIGPLAN Notices (ACM) 12, 2, 1977.

Ledgard H.F. A model for type checking - with an application for Algol 60. Comm. ACM, 15, 11, November 1972, 956-966.

Milner R. A theory of type polymorphism in programming. J. Comp and System Sciences, 17, 1978, 348-375.

Mitchell J.G., Maybury W., Sweet R. Mesa language manual. Version 5.0. CSL-79-3, XEROX PARC, April 1979.

Solomon M. Type definition with parameters. Proc. Fifth ACM Symposium on Principles of Programming Languages, Tucson, January 1978, 31-37.

Reynolds J.C. Toward a theory of type structure. Lecture Notes in Comp. Science 19, Springer Verlag, Proc. Programming Symp., 1974, 408-425.

Tennent R.D. On a new approach to representation-independent data classes. Acta Inform., 1977, 315-324.

TABLE 1

$TE[\![X]\!]env,te = env(X)$

$TE[\![\underline{type}]\!]env,te = \underline{type}$

$TE[\![(D_0,D_1, \ldots,D_n) \underline{in}\ E]\!]env,te =$

$\quad TE[\![(D_1,D_2, \ldots, D_n) \underline{in}\ E]\!] (TD[\![D_0]\!]env,te),te$

$TE[\![() \underline{in}\ E]\!]env,te = TE[\![E]\!]env,te$

$TE[\![\underline{lambda}(X_0;E_0,X_1:E_1, \ldots,X_n:E_n).E_{n+1}]\!]env,te=$

$\quad \underline{if}\ istype(v_i)te_i \quad (i=0,1,\ldots,n)$

$\quad\quad \underline{then}\ \lambda(x_0:v_0,x_1:v_1, \ldots,x_n:v_n).v_{n+1}$

$\quad\quad \underline{else}\ \underline{error}$

$\quad \underline{where}$

$\quad\quad v_i = TE[\![E_i]\!]env_i,te_i \quad i=0,1,\ldots,n+1$

$\quad\quad env_0=env, \quad te_0=te$

$\quad\quad env_i=env_{i-1}\{x_{i-1}/X_{i-1}\}\ ,\ te_i=te_{i-1}\{v_{i-1}/x_{i-1}\} \quad i=1,2,\ldots,n+1$

$TE[\![\underline{func}(X_0:E_0,X_1:E_1, \ldots,X_n:E_n)\rightarrow E_{n+1}]\!]env,te =$

$\quad \underline{if}\ istype(v_i)te_i \quad (i=0,1,\ldots,n) \quad and$

$\quad\quad NF(typeof(NF(v_{n+1})te_{n+1})te_{n+1})te_{n+1} = \underline{type}$

$\quad\quad\quad \underline{then}\ \Lambda(x_0:v_0,x_1:v_1, \ldots,x_n:v_n)\ .\ v_{n+1}$

$\quad\quad\quad \underline{else}\ \underline{error}$

$\quad \underline{where}$

$\quad\quad v_i = TE[\![E_i]\!]env_i,te_i \quad i=0,1,\ldots,n+1$

$\quad\quad env_0=env, \quad te_0=te$

$\quad\quad env_i=env_{i-1}\{x_{i-1}/X_{i-1}\}\ ,\ te_i=te_{i-1}\{v_{i-1}/x_{i-1}\} \quad i=1,2,\ldots,n$

$TE[\![\underline{tconstr}(X_0:E_0,X_1:E_1, \ldots,X_n:E_n)]\!]env,te=$

$\quad \underline{if}\ istype\ (v_i)te_i \quad (i=0,1,\ldots,n)$

$\quad\quad \underline{then}\ \Lambda(x_0:v_0,x_1:v_1, \ldots,x_n:v_n).\underline{type}$

$\quad\quad \underline{else}\ \underline{error}$

$\quad \underline{where}$

$\quad\quad v_i = TE[\![E_i]\!]env_i,te_i \quad i=0,1,\ldots,n$

$\quad\quad env_0 = env, \quad te_0=te$

$\quad\quad env_i=env_{i-1}\{x_{i-1}/X_{i-1}\}, \ te_i=te_{i-1}\{v_{i-1}/X_{i-1}\} \quad i=1,2,\ldots,n$

$TE\,[\![E_{n+1}(E_0,E_1,\ \ldots,E_n)]\!]\ env,te = v_{n+1}(v_0,v_1,\ \ldots,v_n)$

where $v_i = TE[\![E_i]\!]\ env,te \quad (i=0,1,\ldots,n+1)$

$TE[\![\underline{if}\ E_0\ \underline{then}\ E_1\ \underline{else}\ E_2]\!]\ env,te =$

$\langle cond, \Lambda(t:\underline{type}, x_0:\langle boolean, \underline{type}\rangle, x_1:t, x_2:t).t\rangle$

$(typeof(NF(v_1)te)te, v_0, v_1, v_2)$

where $v_i = TE[\![E_i]\!]\ env,te \quad (i=0,1,2)$

$TE[\![\downarrow X]\!]\ env,te = env(\downarrow X)$

$TE[\![\underline{up}(E_1,E_2)]\!]\ env,te =$

$\langle \underline{up}, \Lambda(t_1:\underline{type}, t_2:\underline{type}, x:t_1).t_2\rangle\ (v_2, v'_2, v_1)$

where

$v_i = TE[\![E_i]\!]\ env,te \quad (i=1,2)$

$v'_2 = TE[\![abst(E_2)]\!]\ env,te$

$abst(E) = \{X/\downarrow X\}E$ for each $X \in IDE$

$TE[\![\underline{down}(E_1,E_2)]\!]\ env,te =$

$\langle \underline{down}, \Lambda(t_1:\underline{type}, t_2:\underline{type}, x:t_2).t_1\rangle\ (v_2, v'_2, v_1)$

where

$v_i = TE[\![E_i]\!]\ env,te \quad (i=1,2)$

$v'_2 = TE[\![abst(E_2)]\!]\ env,te$

$abst(E) = \{X/\downarrow X\}E$ for each $X \in IDE$

$D[\![\underline{let}\ X:E_1=E_2]\!]\ env,te =$

$\underline{if}\ istype(v_1)$ and $NF(v_1)te \cong NF(typeof(NF(v_2)te)te)te$

$\underline{then}\ env\{v_2/X\}$

$\underline{else}\ env\{\underline{error}/X\}$

where $v_i = TE[\![E_i]\!]\ env,te \quad (i=1,2)$

TD⟦letrec X:func(X_0:E_0,X_1:E_1, ...,X_n:E_n) → E_{n+1}=E⟧ env,te

if NF(v_1)te ≅ NF(typeof(NF(v_2)te)te)te

then env{v_2/X}

else env{error/X}

where

v_1=TE⟦func(X_0:E_0,X_1:E_1, ...,X_n:E_n) → E_{n+1}⟧env,te

v_2=TE⟦E⟧env{<a,v_1> /X},te

TD⟦new X:type=E⟧env,te =

if NF(typeof(NF(v)te)te)te = type

then env{<a,type> /X} {v/↓X},te

else env{error/X} {error/↓X},te

where v=TE⟦E⟧env {<a,type>/X},te

TD⟦new X:tconstr(X_0:E_0,X_1:E_1, ...,X_n:E_n)=E⟧env,te =

if NF(v_1)te≅NF(typeof(NF(v_2)te)te)te

then env {<a,v_1> /X}{v_2/↓X},te

else env{error/X}{error/↓X},te

where

v_1=TE⟦tconstr(X_0:E_0,X_1:E_1, ...,X_n:E_n)⟧ env,te

v_2=TE⟦E⟧env{<a,v_1> /X},te