TDF Specification

Issue 2.0 Revision 1 (December 1992)

Defence Research Agency
St Andrews Road
Malvern
Worcestershire
WR14 3PS
United Kingdom

Notice to Readers

TDF is a portability technology and an architecture neutral format for expressing software applications which was developed by the United Kingdom's Defence Research Agency (DRA). DRA has demonstrated that the TDF technology can support ANSI C on MIPS®, Intel 386™, VAX ™, SPARC™ and Motorola® 680x0.

Requests for information about TDF should be directed to:

Dr N E Peeling
Defence Research Agency
St Andrews Road
Malvern
Worcestershire
United Kingdom WR14 3PS

Tel +44 684 895314
Fax +44 684 894303
Internet peeling%hermes.mod.uk@relay.mod.uk

This document is for advanced information. It is not necessarily to be regarded as a final or official statement by the Defence Research Agency.

December 1992

Preface

This is Issue 2.0 Revision 1 of the TDF Specification. Issue 2.0 was produced in September 1992 and can be identified by that date in its "Notice to Readers" section.

This revision corrects errors in Issue 2.0 and adds some explanatory material. Except for the changes to *jump_fraction* and *no_of_uses*, the alterations are to correct minor omissions and errors. The changes to *jump_fraction* and *no_of_uses* are made because the more flexible way to use profiling information is by using tokens. The revision neither adds nor subtracts any facilities, nor makes any change in interpretation, except as specified below:-

3.10.5 Effect of applying a null procedure.

3.10.8 SHAPE of result corrected.

3.10.18 Parameters allowed to be marked as visible.

3.10.30 *jump_fraction* obsolete.

3.10.32 Effect when parameter is null.

3.10.34 *jump_fraction* obsolete.

3.10.36 Explanation of effect before any allocation.

3.10.40 *jump_fraction* obsolete.

3.10.41 Effect when parameter is null.

3.10.51 *no_of_uses* obsolete. Legally accessible TAGS defined.

3.10.53 SHAPE will not be BOTTOM.

3.10.64 Second parameter of result shape corrected.

3.10.68 Declared obsolete. This construction was C specific.

3.10.70 Explanation relative to *offset_max*. *jump_fraction* removed.

3.10.74 *jump_fraction* obsolete.

3.10.75 *jump_fraction* obsolete.

3.10.83 Effect of too large shift undefined.

3.10.84 Effect of too large shift undefined.

3.20.5 TOP corrected to { }.

3.22.16 *result* can be a token SORTNAME.

3.32.1 New construction.

1. Introduction

TDF is a porting technology and, as a result, it is a central part of a shrink-wrapping, distribution and installation technology. TDF has been chosen by the Open Software Foundation as the basis of its Architecture Neutral Distribution Format. It was developed by the United Kingdom's Defence Research Agency (DRA). DRA is working with USL to commercialise the TDF technology. TDF is not UNIX specific, although most of the implementation has been done on UNIX.

Software vendors, when they port their programs to several platforms, usually wish to take advantage of the particular features of each platform. That is, they wish the versions of their programs on each platform to be functionally equivalent, but not necessarily algorithmically identical. TDF is intended for porting in this sense. It is designed so that a program in its TDF form can be systematically modifed when it arrives at the target platform to achieve the intended functionality and to use the algorithms and data structures which are appropriate and efficient for the target machine. A fully efficient program, specialised to each target, is a necessity if independent software vendors are to take-up a porting technology.

These modifications are systematic because, on the source machine, programmers work with generalised declarations of the APIs they are using. The declarations express the requirements of the APIs without giving their implementation. The declarations are specified in terms of TDF's "tokens", and the TDF which is produced contains uses of these tokens. On each target machine the tokens are used as the basis for suitable substitutions and alterations. This method is related to abstract data types, but more kinds of object can be abstracted.

Using TDF for porting places extra requirements on software vendors and API designers. Software vendors must write their programs scrupulously in terms of APIs and nothing more. API designers need to produce an interface which can be specialised to efficient data structures and constructions on all relevant machines.

TDF is neutral with respect to the set of languages which has been considered. The design of C, C++, Fortran and Pascal is quite conventional, in the sense that they are sufficiently similar for TDF constructions to be devised to represent them all. These TDF constructions can be chosen so that they are, in most cases, close to the language constructions. Other languages, such as Lisp, are likely to need a few extensions. To express novel language features TDF will probably have to be more seriously extended. But the time to do so is when the feature in question has achieved sufficient stability. Tokens can be used to express the constructs until the time is right. For example, there is a lack of consensus about the best constructions for parallel languages, so that at present TDF would either have to use low level constructions for parallelism or back what might turn out to be the wrong system. In other words it is not yet the time to make generalisations for parallelism as an intrinsic part of TDF.

TDF is neutral with respect to machine architectures. In designing TDF, the aim has been to retain the information which is needed to produce and optimise the machine code, while discarding identifier and syntactic information. So TDF has constructions which are closely related to typical language features and it has an abstract model of memory. We expect that programs expressed in the considered languages can be translated into code which is as efficient as that produced by native compilers for those languages.

Because of these porting features TDF supports shrink-wrapping, distribution and installation. Installation does not have to be left to the end-user; the production of executables can be done anywhere in the chain from software vendor, through dealer and network manager to the end-user.

This document provides English language specifications for each construct in the TDF format and some general notes on various aspects of TDF. It is not a bit-level specification. It is intended for readers who are aware of the general background to TDF but require more detailed information.

2. Structure of TDF

Each piece of TDF program is classified as being of a particular SORT. Some pieces of TDF are LABELS, some are TAGS, some are ERROR_TREATMENTS and so on (to list some of the more transparently named SORTS). The SORTS of the arguments and result of each construct of the TDF format are specified. For instance, *plus* is defined to have three arguments - an ERROR_TREATMENT and two EXPS (short for 'expression') - and to produce an EXP; *goto* has a single LABEL argument and produces an EXP. The specification of the SORTS of the arguments and results of each construct constitutes the syntax of the TDF format. When TDF is represented as a parsed tree it is structured according to this syntax. When it is constructed and read it is in terms of this syntax.

2.1 The Overall Structure of TDF

A separable piece of TDF is called a CAPSULE. A producer generates a CAPSULE; the TDF linker links CAPSULES together to form a CAPSULE; and the final translation process turns a CAPSULE into an object file.

A CAPSULE has an interface to the outside world, its *external_linkage*. This relates the external way of naming things - by identifier - with the internal way - by "linkable entities". Only the very minimum number of things should be named.

Inside the capsule is a number of GROUPS of UNITS. All the UNITS of the same kind are placed together in a GROUP. The GROUPS occur in a specified order within the CAPSULE. Each UNIT contains information about some linkable entities. The linkable entities provide the only method of communication between one UNIT and another, and between a UNIT and the outside world.

TDF allows both the kinds of linkable entity and the kinds of UNIT to be extended. The TDF linker knows about UNITS called "tld2" UNITS. The TDF installers always use the kinds of linkable entity called TOKENS and TAGS, and the kinds of UNIT called "tokdec", "tokdef", "al_tagdef", "linkinfo", "tagdec" and "tagdef". Using the extensibility, extra information can be added to a CAPSULE, which can be used by other programs. As an example of this, diagnostics are represented by some more kinds of entity and GROUP.

The TOKENS are used as internal names of constructions for substitutions. The TAGS are the internal names of run-time values - including procedures. At the CAPSULE level there are TOKENS and TAGS (and each kind of linkable entity) which are available for use by any of the UNITS. Each UNIT has its own linkable entities for its private use. The only communication between separate UNITS is by their reference to the entities at the CAPSULE level.

2.2 Describing the Structure of TDF

The following examples show how TDF constructs are described in this document. The first is the construct *floating*:

v: FLOATING_VARIETY
-> SHAPE

The construct's arguments (one in this case) precede the "->" and the result follows it. Each argument is shown as follows:

name: SORT

The name standing before the colon is for use in the accompanying English description within the specification. It has no other significance.

The example given above indicates that *floating* takes one argument. This argument, *v*, is ofSORT FLOATING_VARIETY. After the "->" comes the SORT of the result of *floating*. It is a SHAPE.

In the case of *floating* the formal description supplies the syntax and the accompanying English text supplies the semantics. However, in the case of some constructs it is convenient to specify more information in the formal section. For example, the specification of the construct *floating_negate* not only states that it has an EXP argument and an EXP result:

> *ov_err*: ERROR_TREATMENT
> *arg1*: EXP FLOATING(*f*)
> -> EXP FLOATING(*f*)

it also supplies additional information about those EXPS. It specifies that these expressions will be floating point numbers of the same size.

Some constructs take a varying number of arguments. *floating_mult* provides an example:

> *ov_err*: ERROR_TREATMENT
> *arg1*:(*i*: 1,*n*) (EXP FLOATING(*f*))[*i*]
> -> EXP FLOATING(*f*)

The number of EXP arguments, *n*, can vary. The index is used to identify the individual components.

Some construct's arguments are optional. This is denoted by the suffix "OPTION" on the name of the argument's SORT:

> *result_shape*: SHAPE
> *arg1*: EXP PROC
> *arg2*: (*i*:1,*n*)(EXP *sh*[*i*])
> *varparam*: EXP *s* OPTION
> -> EXP *result_shape*

varparam is an optional argument to the *apply_proc* construct shown above.

Some constructs' results are governed by the values of their arguments. This is denoted by the "?" formation shown in the specification of the *case* construct shown below:

> *exhaustive*: BOOL
> *control*: EXP INTEGER(*v*)
> *branches*:(*i*: 1, *n*)
> (*branch*[*i*]: LABEL
> *lower*[*i*]: SIGNED_NAT
> *upper*[*i*]: SIGNED_NAT
>)
> -> EXP (*exhaustive* ? BOTTOM : TOP)

If *exhaustive* is true, the resulting EXP has the SHAPE BOTTOM: otherwise it is TOP.

Depending on a TDF-processing tool's purpose not all of some constructs' arguments need necessarily be processed. For instance, installers do not need to process one of the arguments of the *x_cond* constructs. Secondly, standard tools might want to ignore embedded fragments of TDF adhering to some private standard. In these cases it is desirable for tools to be able to skip the irrelevant pieces of TDF: BITSTREAMs and BYTESTREAMs are formations which permit this. In the encoding they are prefaced with information about their length.

Some constructs' arguments are defined as being BITSTREAMs or BYTESTREAMs, even though the constructs specify them to be of a particular SORT. In these cases the argument's SORT is denoted as (eg.):

BITSTREAM FLOATING_VARIETY

The construct in question must have a FLOATING_VARIETY argument, but certain TDF-processing tools may benefit from being able to skip past the argument (which might itself be a very large piece of TDF) without having to read its content.

2.3 Specifying Installer Behaviour

In this document the behaviour of TDF installers is described in a precise manner. Certain words are used with very specific meanings. These are:

- "undefined": means that installers can perform any action, including refusing to translate the program. It can produce code with any effect, meaningful or meaningless.
- "shall": when the phrase "P shall be done" (or similar phrases involving "shall") is used, every installer must perform P.
- "should": when the phrase "P should be done" (or similar phrase involving "should") is used, installers are advised to perform P, and producer writers may assume it will be done if possible. This usage generally relates to optimisations which are recommended.
- "will": when the phrase "P will be true" (or similar phrases involving "will") is used to describe the composition of a TDF construct, the installer may assume that P holds without having to check it. If, in fact, a producer has produced TDF for which P does not hold, the effect is undefined.
- "target-defined": means that behaviour will be defined, but that it varies from one target machine to another. Each target installer shall define everything which is said to be "target-defined".

2.4 Properties of Installers

All installers must implement all of the constructions of TDF. There are some constructions where the installers may impose limits on the ranges of values which are implemented. In these cases the description of the installer must specify these limits. Implementations without such limits are possible, and should always be preferred.

3. Specification of TDF Constructs

Each class of TDF construct - or "SORT" - is specified in alphabetical order.

3.1 AL_TAG

AL_TAGS name ALIGNMENTS.

3.1.1 al_tag_apply_token

> *token_value*: TOKEN
> *token_args*: BITSTREAM
> -> AL_TAG

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an AL_TAG.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.1.2 make_al_tag

> *al_tagno*: TDFINT
> -> AL_TAG

make_al_tag construct an AL_TAG identified by *al_tagno*.

3.2 AL_TAGDEF

An AL_TAGDEF gives the definition of an AL_TAG for incorporation into a UNIT.

3.2.1 make_al_tagdef

> *t*: TDFINT
> *a*: ALIGNMENT
> -> AL_TAGDEF

The AL_TAG identified by *t* is defined to stand for the ALIGNMENT *a*. All the AL_TAGDEFS in a CAPSULE must be considered together as a set of simultaneous equations defining ALIGNMENT values for the AL_TAGS. No order is imposed on the definitions.

In any particular CAPSULE the set of equations may be incomplete, but a CAPSULE which is being translated into code will have a set of equations which defines all the AL_TAGS which it uses.

Simultaneous equations defining ALIGNMENTS can always be solved, since the only significant combining operation is *unite_alignments*, which is set union.

3.3 ALIGNMENT

An ALIGNMENT is a set of basic information about the SHAPES of values which are pointed at. It is used to express the information needed for the parameters of the POINTER and OFFSET constructions.

The possible values of the elements in such a set are *proc*, *local_label_value*, *pointer*, *offset*, all VARIETIES, all FLOATING_VARIETIES and all BITFIELD_VARIETIES.

3.3.1 alignment_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> ALIGNMENT

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an ALIGNMENT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.3.2 alignment_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM ALIGNMENT
e2: BITSTREAM ALIGNMENT
-> ALIGNMENT

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.3.3 alignment

sha: SHAPE
-> ALIGNMENT

The *alignment* construct is defined as follows.

If *sha* is LOCAL_LABEL_VALUE then the resulting ALIGNMENT is {*local_label_value*} - the singleton set containing *local_label_value*.

If *sha* is PROC then the resulting ALIGNMENT is {*proc*}.

If *sha* is INTEGER(*v*) then the resulting ALIGNMENT is {*v*}.

If *sha* is FLOATING_VARIETY(*v*) then the resulting ALIGNMENT is {*v*}.

If *sha* is BITFIELD(*v*) then the resulting ALIGNMENT is {*v*}.

If *sha* is TOP the resulting ALIGNMENT is { } - the empty set.

If *sha* is BOTTOM the resulting ALIGNMENT is undefined.

If *sha* is POINTER(*x*) or OFFSET(*x*, *y*) then the resulting ALIGNMENT is {*pointer*} or {*offset*} respectively.

If *sha* is NOF(*n*, *s*) the resulting ALIGNMENT is *alignment*(*s*).

If *sha* is COMPOUND(EXP OFFSET(x, y)) then the resulting ALIGNMENT is x.

3.3.4 obtain_al_tag

at: AL_TAG
    -> ALIGNMENT

obtain_al_tag produces the ALIGNMENT with which the AL_TAG *at* is bound.

3.3.5 unite_alignments

a1: ALIGNMENT
a2: ALIGNMENT
    -> ALIGNMENT

unite_alignments produces the alignment at which all the members of the ALIGNMENT sets *a1* and *a2* can be placed - in other words the ALIGNMENT set which is the union of *a1* and *a2*.

3.4 BITFIELD_VARIETY

These describe runtime bitfield values. The intention is that these values are usually kept in memory locations which need not be aligned on addressing boundaries.

There is no limit on the size of bitfield values in TDF, but an installer may specify limits. Every installer shall implement bitfields up to 32 bits, both signed and unsigned. A statement of any such limits shall be part of the specification of an installer. Installers are encouraged not to place any limits on the size of bitfields.

3.4.1 bfvar_apply_token

token_value: TOKEN
token_args: BITSTREAM
    -> BITFIELD_VARIETY

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give a BITFIELD_VARIETY.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.4.2 bfvar_cond

control: EXP INTEGER(v)
e1: BITSTREAM BITFIELD_VARIETY
e2: BITSTREAM BITFIELD_VARIETY
    -> BITFIELD_VARIETY

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.4.3 bfvar_bits

issigned: BOOL
bits: NAT
-> BITFIELD_VARIETY

bfvar_bits constructs a BITFIELD_VARIETY describing a pattern of *bits* bits. If *issigned* is *true*, the pattern is considered to be a twos-complement signed number: otherwise it is considered to be unsigned.

3.5 BITSTREAM

A BITSTREAM consists of an encoding of any number of bits. This encoding is such that any program reading TDF can determine how to skip over it. To read it meaningfully extra knowledge of what it represents may be needed.

A BITSTREAM is used, for example, to supply parameters in a TOKEN application. If there is a definition of this TOKEN available, this will provide the information needed to decode the bitstream.

3.6 BOOL

A BOOL is a piece of TDF which can take two values, *true* or *false*.

3.6.1 bool_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> BOOL

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give a BOOL.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.6.2 bool_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM BOOL
e2: BITSTREAM BOOL
-> BOOL

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.6.3 false

-> BOOL

false produces a false BOOL.

3.6.4 true

-> BOOL

true produces a true BOOL.

3.7 BYTESTREAM

A BYTESTREAM is analogous to a BITSTREAM, but is encoded to permit fast copying.

3.8 CAPSULE

A CAPSULE is an independent piece of TDF. There is only one construction, *make_capsule*.

3.8.1 make_capsule

prop_names: (*q*: 1, *n_eqn*) TDFIDENT(*nc*[*q*])[*q*]
capsule_linking: (*r*: 1, *n_link*) (*sn*[*r*] : TDFIDENT(*nl*[*r*]), *no*[*r*] : TDFINT)

external_linkage: (*t*: 1, *n_link*)((*s*[*t*]: 1, *el*[*t*]) LINKEXTERN[*s*[*t*]])
units: (*u*: 1, *n_eqn*) ((*v*[*u*]: 1, *e*[*u*]) UNIT[*v*[*u*]])
-> CAPSULE

make_capsule brings together UNITS, information about internal and external naming and information to assist tools which process TDF.

One *prop_name* corresponds to each group of UNITS. The *prop_names* indicate the kind of information which their matching UNITS contain. Groups of UNITS formed from PROPS constructed using *make_al_tagdefs* are denoted by the TDFIDENT "al_tagdefs", *make_tagdecs* by "tagdecs", *make_tagdefs* by "tagdefs", *make_tokdecs* by "tokdecs" and *make_tokdefs* by "tokdefs". Not all of these need to be present in a CAPSULE. The groups of UNITS occur in *units* in the same order as the corresponding TDFIDENTS in *prop_names*.

It is intended that new kinds of PROPS can be added to the standard in a purely additive fashion, either to form a new standard or for private purposes.

capsule_linking indicates how many TDFIDENTS are employed at the CAPSULE level for identifying each kind of thing. Such identifications are made either to allow something to be externally named, or to permit two UNITS to refer to the same thing (or for both purposes). For example, (sn:"token", no:6) signifies that six TOKENS are used at the CAPSULE level. These TDFINTS which enable such linking are known as linkable entities.

It is intended that new kinds of linkable entity can be added to the standard in a purely additive fashion, either to form a new standard or for private purposes.

external_linkage provides a list of lists of LINKEXTERNS, each outer list corresponding to a different kind of linkable entity and all appearing in the same order as in *capsule_linking*. These LINKEXTERNS specify the associations between the names to be used outside the CAPSULE and the names by which the UNITS make objects available within the CAPSULE.

Lastly, the *units* provide the contents of the CAPSULE. In this list of lists, UNITS with the same PROPS are grouped together.

The UNITS may refer to linkable entities at the CAPSULE level, and thus different UNITS may refer to the same linkable entity. Such correspondences are set up in a UNIT's LINKS.

3.9 ERROR_TREATMENT

These values describe the way to handle various forms of error which can occur during the evaluation of operations.

3.9.1 errt_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> ERROR_TREATMENT

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an ERROR_TREATMENT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.9.2 errt_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM ERROR_TREATMENT
e2: BITSTREAM ERROR_TREATMENT
-> ERROR_TREATMENT

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.9.3 error_jump

lab: LABEL
-> ERROR_TREATMENT

error_jump produces an ERROR_TREATMENT which requires that control be passed to the LABEL *lab* if it is invoked. The LABEL will be in scope.

3.9.4 ignore

-> ERROR_TREATMENT

ignore is an ERROR_TREATMENT which means, if it is invoked, that the error will be handled in a way which is defined in the construct in which the error occurred.

3.9.5 impossible

-> ERROR_TREATMENT

impossible is an ERROR_TREATMENT which means that this error will not occur in the construct concerned.

3.10 EXP

EXPS are pieces of TDF which are translated into program. EXP is by far the richest SORT. There are few primitive EXPS: most of the constructions take arguments which are a mixture of EXPS and other SORTS. There are constructs delivering EXPS that correspond to the declarations, program structure, procedure calls, assignments, pointer manipulation, arithmetic operations, tests etc. of programming languages.

3.10.1 exp_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> EXP *x*

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an EXP.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.10.2 exp_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM EXP *x*
e2: BITSTREAM EXP *y*
-> EXP *x* or EXP *y*

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.10.3 add_to_ptr

arg1: EXP POINTER(*x*)
arg2: EXP OFFSET(*y*, *z*)
-> EXP POINTER(*z*)

arg1 and *arg2* are evaluated and added to produce the result. The result is derived from the pointer delivered by *arg1*.

x will include *y*.

3.10.4 and

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(*v*)

The arguments are evaluated producing integer values of the same VARIETY, *v*. These values are "anded". The result is delivered with the same SHAPE as the arguments.

3.10.5 apply_proc

result_shape: SHAPE

arg1: EXP PROC
arg2: (*i*:1,*n*)(EXP *sh*[*i*])
varparam: EXP *s* OPTION
-> EXP *result_shape*

arg1, *arg2* (and *varparam* if present) are evaluated. The procedure is applied to the parameters, *arg2*. At this call it will deliver a result of SHAPE *result_shape*.

The constituents of *arg2* will have the SHAPES specified by the formal parameters of the procedure which is being called. The significance of *varparam* is explained in the specification of *make_proc*.

If *arg1* delivers a null procedure the effect is undefined.

In the canonical order of evaluation the operations of the procedure body are started at the time when the *apply_proc* is evaluated, and are all completed before any further operations are evaluated.

3.10.6 assign

arg1: EXP POINTER(*x*)
arg2: EXP *y*
-> EXP TOP

The value produced by *arg2* will be put in the space indicated by *arg1*.

x will include ALIGNMENT(*y*).

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

y may be TOP, and if so translators are recommended to produce no code if possible.

If the value delivered by *arg1* is a null pointer the effect is undefined.

If a jump out of *arg2* occurs, for example because of an ERROR_TREATMENT, the contents of the space indicated by *arg1* are undefined.

3.10.7 assign_to_volatile

arg1: EXP POINTER(*x*)
arg2: EXP *y*
-> EXP TOP

The value produced by *arg2* will be put in the space indicated by *arg1*. This operation shall always be performed, it shall not be optimised out.

x will include ALIGNMENT(*y*).

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

y may be TOP.

If the value delivered by *arg1* is a null pointer the effect is undefined.

If a jump out of *arg2* occurs, for example because of an ERROR_TREATMENT, the contents of the space indicated by *arg1* are undefined.

3.10.8 case

exhaustive: BOOL

control: EXP INTEGER(*v*)
branches:(*i*: 1, *n*)
(*branch*[*i*]: LABEL
lower[*i*]:SIGNED_NAT,
upper[*i*]: SIGNED_NAT
)
-> EXP (exhaustive ? BOTTOM : TOP)

control is evaluated to produce an integer value, *c*. Then *c* is tested to see if it lies inclusively between *lower*[*i*] and upper[*i*], for each *i*. If this tests succeeds for some *i*, control passes to the label *branch*[*i*]. If *c* lies between no pair the construct delivers a value of SHAPE TOP. The order in which the comparisons are made is undefined.

The sets of SIGNED_NATS will be disjoint.

If *exhaustive* is true the value delivered by *control* will lie between one of the lower/upper pairs.

3.10.9 change_bitfield_to_int

x: VARIETY
arg1: EXP BITFIELD *v*
-> EXP INTEGER *x*

arg1 when evaluated gives a value of SHAPE BITFIELD *v*, which is converted to INTEGER *x* and delivered.

If the value is not representable as a value of the result SHAPE the effect is undefined.

3.10.10 change_floating_variety

r: FLOATING_VARIETY
ov_err: ERROR_TREATMENT
arg1: EXP FLOATING(*f*)
-> EXP FLOATING(*r*)

arg1 is evaluated and will produce an floating point value, *fp*. The value *fp* is delivered, changed to the representation of the FLOATING_VARIETY *r*.

If *fp* is not contained in the FLOATING_VARIETY being used to represent *r*, an overflow error is handled according to *ov_error*. If *ov_err* is *ignore*, the effect is undefined.

3.10.11 change_int_to_bitfield

x: BITFIELD_VARIETY
arg1: EXP INTEGER *v*
-> EXP BITFIELD *x*

arg1 when evaluated gives a value of SHAPE INTEGER *v*, which is converted to BITFIELD *x* and delivered.

If the value is not representable as a value of the result SHAPE the effect is undefined.

3.10.12 change_variety

r: VARIETY

arg1: EXP INTEGER(*v*)
-> EXP INTEGER(*r*)

arg1 is evaluated and will produce an integer value, *a*. The value *a* is delivered, changed to the representation of the VARIETY *r*.

If *a* is not contained in the VARIETY being used to represent *r*, the following rules are applied.

Let the variety representing *r* be (*low_r*, *high_r*). Let the variety representing *v* be (*low_v*, *high_v*).

If *a* is negative, *low_r* is zero, *low_v* is negative and (*high_r*-*low_r*) is greater than or equal to (*high_v*-*low_v*), then the result is *a*+*high_r*+1.

If *low_r* is zero and (*high_r*-*low_r*) is less than (*high_v*-*low_v*) then the result is *a* M1 (*high_r*+1).

3.10.13 component

sha: SHAPE
arg1: EXP COMPOUND(EXP OFFSET(*x*, *y*))
arg2: EXP OFFSET(*x*, *alignment*(*sha*))
-> EXP *sha*

arg1 is evaluated to produce a COMPOUND value. The component of the compound value at the OFFSET given by *arg2* is delivered. This will have SHAPE *sha*.

3.10.14 concat_nof

arg1: EXP NOF(*n*, *s*)
arg2: EXP NOF(*m*, *s*)
-> EXP NOF(*n*+*m*, *s*)

arg1 and *arg2* are evaluated and their results concatenated.

3.10.15 conditional

alt_label_intro: LABEL

first: EXP *x*
alt: EXP *z*
-> EXP (*x* LUB *z*)

first is evaluated. If *first* produces a result, *f*, this value is delivered as the result of the whole construct, and *alt* is not evaluated.

If *goto*(*alt_label*) or any other jump to *alt_label* is obeyed during the evaluation of *first*, the the evaluation of *first* will stop, *alt* will be evaluated and its result delivered as the result of the construction.

The lifetime of *alt_label* is the evaluation of *first*. Note that *alt_label* is not available within *alt*.

The actual order of evaluation of the constituents shall be indistinguishable in all observable effects (apart from time) from evaluating all the obeyed parts of *first* before any obeyed part of *alt*. Note that this specifically includes any defined error handling.

3.10.16 contents

s: SHAPE

arg1: EXP POINTER(*x*)
-> EXP *s*

A value of SHAPE *s* will be extracted from the start of the space indicated by the pointer, and this is delivered.

x will include *alignment*(*s*).

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

s may be TOP, and if so most translators will produce no code.

If the value delivered by *arg1* is a null pointer the effect is undefined.

3.10.17 contents_of_volatile

s: SHAPE
arg1: EXP POINTER(*x*)
-> EXP *s*

A value of SHAPE *s* will be extracted from the start of the space indicated by the pointer, and this is delivered. This operation shall always be performed, it shall not be optimised away.

x will include *alignment*(*s*).

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

s may be TOP.

If the value delivered by *arg1* is a null pointer the effect is undefined.

3.10.18 current_env

-> EXP POINTER(*frame_alignment*)

A value of SHAPE POINTER(*frame_alignment*) is created and delivered. It gives access to the variables, identities and parameters in the current procedure activation which are declared as *visible*.

If an OFFSET produced by *env_offset* is added to a POINTER produced by *current_env* from an activation of the procedure which contains the declaration of the TAG used by *env_offset*, then the result is an original POINTER, notwithstanding the normal rules for *add_to_ptr*.

If an OFFSET produced by *env_offset* is added to such a pointer from an inappropriate procedure the effect is undefined.

3.10.19 div1

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* D1 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v* the effect is undefined.

If *b* is zero the effect is undefined.

Producers may assume that shifting and div1 by a power of two yield equally good code.

See the section "Division and Modulus" for the definitions of D1, D2, M1 and M2.

3.10.20 div2

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* D2 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v* the effect is undefined.

If *b* is zero the effect is undefined.

Producers may assume that shifting and div2 by a power of two yield equally good code if the lower bound of *v* is zero.

See the section "Division and Modulus" for the definitions of D1, D2, M1 and M2.

3.10.21 env_offset

t: TAG *x*
-> EXP OFFSET(*frame_alignment*, *x*)

t will be the tag of a *variable*, *identify* or procedure parameter with the *visible* property.

If it is a *variable* or a procedure parameter, the result is the OFFSET of the space of the given variable, within any procedure environment which derives from the procedure containing the declaration of the variable, relative to its environment pointer.

If it is an *identify*, the result will be an OFFSET of space which holds the value. This pointer will not be used to alter the value.

The ALIGNMENT, *frame_alignment*, is the set union of all the ALIGNMENTS which can be produced by *alignment* from any SHAPE.

3.10.22 fail_installer

message: TDFSTRING(*k*, *n*)
-> EXP BOTTOM

Any attempt to use this operation to produce code will result in a failure of the installation process. *message* will give information about the reason for this failure which should be passed to the installation manager.

3.10.23 float_int

f: FLOATING_VARIETY
ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
-> EXP FLOATING(*f*)

arg1 is evaluated to produce an integer value, which is converted to the representation of *f* and delivered. Any rounding necessary is target-defined.

If the integer value cannot be expressed in the representation of *f*, an overflow is caused and handled by *ov_err*.

If *ov_err* is *ignore*, overflow behaviour is undefined.

3.10.24 floating_abs

ov_err: ERROR_TREATMENT
arg1: EXP FLOATING(*f*)
-> EXP FLOATING(*f*)

arg1 is evaluated and will produce a floating point value, *a*, of the FLOATING_VARIETY, *f*. The absolute value of *a* is delivered as the result of the construct, with the same SHAPE as the argument.

If the result cannot be expressed in the FLOATING_VARIETY being used to represent *f*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *ov_err* is *ignore* the effect of overflow is undefined.

3.10.25 floating_div

ov_err: ERROR_TREATMENT
arg1: EXP FLOATING(*f*)
arg2: EXP FLOATING(*f*)
-> EXP FLOATING(*f*)

arg1 and *arg2* are evaluated and will produce floating point values, *a* and *b*, of the same FLOATING_VARIETY, *f*. The value *a*/*b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the FLOATING_VARIETY being used to represent *f*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *ov_err* is *ignore* the effect of overflow is undefined.

If *b* is zero the effect is undefined.

3.10.26 floating_minus

ov_err: ERROR_TREATMENT
arg1: EXP FLOATING(*f*)
arg2: EXP FLOATING(*f*)
-> EXP FLOATING(*f*)

arg1 and *arg2* are evaluated and will produce floating point values, *a* and *b*, of the same FLOATING_VARIETY, *f*. The value *a*-*b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the FLOATING_VARIETY being used to represent *f*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *ov_err* is *ignore* the effect of overflow is undefined.

3.10.27 floating_mult

ov_err: ERROR_TREATMENT

arg1: (*i*: 1, *n*)(EXP FLOATING (*f*)[*i*])
-> EXP FLOATING(*f*)

The arguments, *arg1*, are evaluated producing floating point values, *a*[*i*], all of the same FLOATING_VARIETY, *f*. These values are multiplied in any order and the result of this multiplication is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result, or any partial result, cannot be expressed in the FLOATING_VARIETY being used to represent *f*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *ov_err* is *ignore* the effect of overflow is undefined.

Note that separate floating_mult operations cannot in general be combined, because rounding errors need to be controlled. The reason for allowing *floating_mult* to take a variable number of arguments is to make it possible to specify that a number of multiplications can be re-ordered.

If *n* is zero the result is one, if *n* is one the result is the value of the single argument.

3.10.28 floating_negate

ov_err: ERROR_TREATMENT
arg1: EXP FLOATING(*f*)
-> EXP FLOATING(*f*)

arg1 is evaluated and will produce a floating point value, *a*, of the FLOATING_VARIETY, *f*. The value -*a* is delivered as the result of the construct, with the same SHAPE as the argument.

If the result cannot be expressed in the FLOATING_VARIETY being used to represent *f*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *ov_err* is *ignore* the effect of overflow is undefined.

3.10.29 floating_plus

ov_err: ERROR_TREATMENT
arg1: (*i*: 1, *n*)(EXP FLOATING (*f*)[*i*])
-> EXP FLOATING(*f*)

The arguments, *arg1*, are evaluated producing floating point values, *a*[*i*], all of the same FLOATING_VARIETY, *f*. These values are added in any order and the result of this addition is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result, or any partial result, cannot be expressed in the FLOATING_VARIETY being used to represent *f*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *ov_err* is *ignore* the effect of overflow is undefined.

Note that separate floating_plus operations cannot in general be combined, because rounding errors need to be controlled. The reason for allowing *floating_plus* to take a variable number of arguments is to make it possible to specify that a number of multiplications can be re-ordered.

If *n* is zero the result is zero, if *n* is one the result is the value of the single argument.

3.10.30 floating_test

jump_fraction: NAT_OPTION
nt: NTEST
dest: LABEL

arg1: EXP FLOATING(*f*)
arg2: EXP FLOATING(*f*)
-> EXP TOP

arg1 and *arg2* are evaluated and will produce floating point values, *a* and *b*, of the same FLOATING_VARIETY, *f*. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

jump_fraction is obsolete and will be ignored.

3.10.31 goto

dest: LABEL
-> EXP BOTTOM

Control passes to the EXP labelled *dest*. This construct will only be used where *dest* is in scope.

3.10.32 goto_local_lv

arg1: EXP LOCAL_LABEL_VALUE
-> EXP BOTTOM

arg1 is evaluated. The label from which the value delivered by *arg1* was created will be within its lifetime and this construction will be obeyed in the same activation of the same procedure as the creation of the LABEL_VALUE by *make_local_lv*. Control passes to this activation of this LABEL.

If *arg1* delivers a null LOCAL_LABEL_VALUE the effect is undefined.

3.10.33 identify

visible: BOOL
name_intro: TAG *x*
definition: EXP *x*

body: EXP *y*
-> EXP *y*

definition is evaluated to produce a value, *v*. Then *body* is evaluated. During this evaluation, *v* is bound to *name_intro*. This means that inside *body* an evaluation of *obtain_tag*(*name_intro*) will produce the value, *v*.

The value delivered by *identify* is that produced by *body*.

The TAG given for *name_intro* will not be reused within the current UNIT. No rules for the hiding of one TAG by another are given: this will not happen. The lifetime of *name_intro* is the evaluation of *body*.

If *visible* is true, it means that the value must not be aliased while the procedure containing this declaration is not the current procedure. Hence if there are any copies of this value they will need to be refreshed when the procedure is returned to. The easiest implementation when *visible* is true may be to keep the value in memory, but this is not a necessary requirement.

The order in which the constituents of *definition* and *body* are evaluated shall be indistinguishable in all observable effects (apart from time) from completely evaluating *definition* before starting *body*. See the note about order in *sequence*.

3.10.34 integer_test

> *jump_fraction*: NAT_OPTION
> *nt*: NTEST
> *dest*: LABEL
> *arg1*: EXP INTEGER(*v*)
> *arg2*: EXP INTEGER(*v*)
> -> EXP TOP

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

jump_fraction is obsolete and will be ignored.

3.10.35 labelled

> *placelabs_intro*: (*i*: 1, *n*)(*place_label*[*i*]: LABEL)
>
> *starter*: EXP *x*
> *places*: (*j*: 1, *n*)(*place*[*j*]: EXP *z*[*j*])
> -> EXP *w*

starter is evaluated. If its evaluation runs to completion producing a value, then this is delivered as the result of the whole construction. If a *goto*(*place_label*[*k*]) or any other jump to *place_label*[*k*] is evaluated, then the evaluation of *starter* stops and *place*[*k*] is evaluated. In the canonical ordering all the operations which are evaluated from *starter* are completed before any from *place*[*k*] is started. If *place*[*k*] produces a result this is the result of the construction.

If a jump to any of the *place_labels* is obeyed then evaluation continues similarly. Such jumping may continue indefinitely, but if any *place* terminates, then the value it produces is the value delivered by the construction.

The SHAPE *w* is the LUB of *x* and all the *z*[*j*].

The actual order of evaluation of the constituents shall be indistinguishable in all observable effects (apart from time) from that described above. Note that this specifically includes any defined error handling.

The lifetime of each of the LABELS, *place_label*[*i*], is the evaluation of *starter* and all the *place*[*j*].

3.10.36 last_local

> *x*: ALIGNMENT
> -> EXP POINTER(*x*)

If the last use of *local_alloc* in the current activation of the current procedure was after the last use of *local_free* or *local_free_all*, then the value returned is the last POINTER allocated with *local_alloc*. This POINTER will have been created with the ALIGNMENT *x*.

If the last use of *local_free* in the current activation of the current procedure was after the last use of *local_alloc*, then the result is the POINTER last allocated which is still active.

If the last use of *local_free_all* in the current activation of the current procedure was after the last use of *local_alloc*, or if there has been no use of *local_alloc* in the current activation of the current procedure, then the result is such that a subsequent use of *local_free* applied to this result will be equivalent to *local_free_all*. In these circumstances, the effect of applying *contents* or *assign* to the result is undefined.

3.10.37 local_alloc

arg1: EXP OFFSET(x, y)
    -> EXP POINTER(x)

The *arg1* expression is evaluated and space is allocated sufficient to hold a value of the given size. The result is an original pointer to this space.

The initial contents of the space are undefined.

This allocation is as if on the stack of the current procedure, and the lifetime of the pointer ends when the current activation of the current procedure ends. Any use of the pointer thereafter is undefined.

The uses of *local_alloc* within the procedure are ordered dynamically as they occur, and this order affects the meaning of *local_free* and *last_local*.

Note that if a procedure which uses *local_alloc* is inlined, it may be necessary to use *local_free* to get the correct semantics.

3.10.38 local_free

p: EXP POINTER(x)
    -> EXP TOP

The pointer, *p*, will be an original pointer to space allocated by *local_alloc* within the current call of the current procedure. All spaces allocated after it by *local_alloc* will no longer be used.

Any subsequent use of pointers to the spaces no longer used will be undefined.

3.10.39 local_free_all

    -> EXP TOP

Each space allocated by *local_alloc* within the current call of the current procedure will no longer be used and may be reused.

Any use of a pointer to space allocated before this operation within the current call of the current procedure is undefined.

Note that if a procedure which uses *local_free_all* is inlined, it may be necessary to use *local_free* to get the correct semantics.

3.10.40 local_lv_test

jump_fraction: NAT_OPTION
nt: NTEST
dest: LABEL
arg1: EXP LOCAL_LABEL_VALUE
arg2: EXP LOCAL_LABEL_VALUE
    -> EXP TOP

arg1 and *arg2* are evaluated and will produce LOCAL_LABEL_VALUES, *a* and *b*. These values are compared using *nt*. The only permitted values of *nt* are *equal* and *not_equal*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

Two LOCAL_LABEL_VALUES are equal if they were both made from the same label with *make_local_lv* or if they were both made with *make_null_local_lv*. Otherwise they are unequal.

jump_fraction is obsolete and will be ignored.

3.10.41 long_jump

> *arg1*: EXP POINTER(*frame_alignment*)
> *arg2*: EXP LOCAL_LABEL_VALUE
> -> EXP BOTTOM

The frame produced by *arg1* is reinstated as the current procedure. This frame will still be active. Evaluation recommences at the label given by *arg2*. This operation will only be used during the lifetime of that label.

Only TAGS declared to be visible will be defined at the re-entry.

If *arg2* delivers a null LOCAL_LABEL_VALUE the effect is undefined.

3.10.42 make_compound

> *arg1*: EXP OFFSET(*base*, *y*)
> *arg2*: (*i*: 1, 2 * *n*)(EXP *x*[*i*] : *s*[*i*])
> -> EXP COMPOUND(EXP OFFSET(*base*, *y*))

The components *x*[2 * *k*] are values which are to be placed at OFFSETS given by *x*[2 * *k* - 1]. These OFFSETS will be constants. *n* may be zero.

The OFFSET *x*[2 * *k* - 1] will have the SHAPE OFFSET(*base*, *alignment*(*s*[2 * *k*])).

Any *s*[2 * *k*] may be TOP.

3.10.43 make_floating

> *f*: FLOATING_VARIETY
> *rm*: ROUNDING_MODE
> *sign*: BOOL
> *mantissa*: TDFSTRING(*k*, *n*)
> *base*: NAT
> *exponent*: SIGNED_NAT
> -> EXP FLOATING(*f*)

mantissa will be a STRING of ASCII characters, each of which is either ASCII's point symbol or is greater than or equal to ASCII's zero symbol. Each character, *c*, other than a point symbol will represent the digit *c*-48. The point symbol will not occur more than once.

The BOOL *sign* determines the sign of the result, if true the result will be positive, if false, negative.

A floating point number, *mantissa**(*base*^*exponent*) is created and rounded to the representation of *f* as specified by *rm*. *rm* will not be *round_as_state*.

3.10.44 make_int

> *v*: VARIETY
> *value*: SIGNED_NAT
> -> EXP INTEGER(*v*)

An integer value is delivered of which the value is given by *value*, and the VARIETY by *v*. The SIGNED_NAT *value* will lie between the bounds of *v*.

3.10.45 make_local_lv

lab: LABEL
 -> EXP LOCAL_LABEL_VALUE

A LOCAL_LABEL_VALUE *lv* is created and delivered. It can be used as an argument to *goto_local_lv*. If and when *goto_local_lv* is evaluated with *lv* as an argument, control will pass to *lab*. This will only occur when the current activation of the current procedure is the same as that which was active at the time of the evaluation of the *make_label_value* which created *lv*.

3.10.46 make_nof

arg1: (i: 1, n)($x[i]$: EXP s)
 -> EXP NOF(n, s)

Creates an array of n values of SHAPE s, containing the given values produced by the $x[i]$ in the same order.

n may be zero.

3.10.47 make_nof_int

v: VARIETY
str: TDFSTRING(k, n)
 -> EXP NOF(n, INTEGER(v))

An NOF INTEGER is delivered. The conversions are carried out as if the elements of *str* were INTEGER(*var_limits*(0,2^*k*-1)).

3.10.48 make_null_local_lv

 -> EXP LOCAL_LABEL_VALUE

Makes a null LOCAL_LABEL_VALUE which can be detected by *local_lv_test*. The effect of *goto_local_lv* applied to this is undefined.

3.10.49 make_null_proc

 -> EXP PROC

A null PROC is created and delivered. The null PROC may be tested for by using *proc_test*. The effect of using it as the first argument of *apply_proc* is undefined.

3.10.50 make_null_ptr

a: ALIGNMENT
 -> EXP POINTER(a)

A null POINTER(a) is created and delivered. The null POINTER may be tested for by *pointer_test*.

3.10.51 make_proc

no_of_uses: NAT_OPTION
params_intro: (*i*: 1,*n*)(*sh*[*i*]: SHAPE, *visible*[*i*]: BOOL, *tg*[*i*]: TAG POINTER(*alignment*(*sh*)))
vartag: (*a*: ALIGNMENT, *vtg*: TAG POINTER(*a*)) OPTION

body: EXP BOTTOM
-> EXP PROC

Evaluation of *make_proc* delivers a PROC. When this procedure is applied to parameters using *apply_proc*, space is allocated to hold values of the parameters *tg*[*i*] (and *vartag* if present). The values produced by the actual parameters are used to initialise these spaces. Then *body* is evaluated. During this evaluation *tg*[*i*] and *vartag* (if present) are bound to original POINTERS to these spaces. The lifetime of *tg*[*i*] and *vartag* is the evaluation of *body*.

If *vartag* is present then all uses of *apply_proc* which have the effect of calling this procedure will have their *varparam* option present and the ALIGNMENT of the SHAPE of the *varparam* will be included by *a*. The components of the actual *varparam* will be accessed in *body* by using POINTER and OFFSET arithmetic, and the SHAPE of the actual parameter will correspond to that required by the pointer and offset arithmetic in that application.

The SHAPE of *body* will be BOTTOM. *n* may be zero.

The TAGS used for *tg*[*i*] will not be reused within the current UNIT.

When the procedure is called (by *apply_proc*) the actual parameters supplied in the call will correspond one-to-one with the formal parameters, *tg*[*i*]. These actual parameters will act as initialising values as if the actual and formal parameters formed a variable declaration with *body* as its body.

sh[*i*] specifies the SHAPE of the *ith* parameter.

visible[*i*] specifies whether the *ith* parameter is to be visible in the same sense as a variable declaration.

In *body* the only TAGS which may be used as an argument of *obtain_tag* are those which are declared by *identify* or *variable* constructions in *body* and which are in scope, or TAGS which are declared by *make_id_tagdef* and *make_var_tagdef* or are *tg*[*i*] or *vtg*.

no_of_uses is obsolete and will be ignored.

3.10.52 make_top

-> EXP TOP

make_top delivers a value of SHAPE TOP (ie. void).

3.10.53 make_value

s: SHAPE
-> EXP *s*

This EXP creates some value with the representation of the SHAPE *s*. Although this value will have the correct form, any operation on the value has undefined effect.

Installers will usually be able to implement this operation by producing no code.

The SHAPE *s* will not be BOTTOM.

3.10.54 minus

ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The difference *a-b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by *ov_err*.

For the definition of *ignore* see "Overflow and Integers"

3.10.55 move_some

no_overlap: BOOL
arg1: EXP POINTER *x*
arg2: EXP POINTER *y*
arg3: EXP OFFSET(*z*, *t*)
-> EXP TOP

The arguments are evaluated to produce *p1*, *p2*, and *sz* respectively. A quantity of data measured by *sz* in the space indicated by *p1* is moved to the space indicated by *p2*.

x will include *z* and *y* will include *z*.

If *no_overlap* is true, the source and destination spaces will not overlap. If it is false they may overlap. If the spaces overlap and *no_overlap* is false, translators shall implement the operation as if the data had been moved to a fresh space from *p1*, and copied from there into the destination space.

If the spaces of size *sz* to which *p1* and *p2* point do not lie entirely within the spaces indicated by the original pointers from which they are derived, the effect of the operation is undefined.

If the value delivered by *arg1* or *arg2* is a null pointer the effect is undefined.

3.10.56 mult

ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The product *a**b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by *ov_err*.

For the definition of *ignore* see "Overflow and Integers"

3.10.57 n_copies

n: NAT
arg1: EXP *x*
-> EXP NOF(*n*, *x*)

arg1 is evaluated and an NOF value is delivered which contains *n* copies of this value. *n* can be zero or one or greater.

Producers are encouraged to use *n_copies* to initialise arrays of known size, both statically and dynamically.

3.10.58 negate

ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
    -> EXP INTEGER(*v*)

arg1 is evaluated and will produce an integer value, *a*. The value -*a* is delivered as the result of the construct, with the same SHAPE as the argument.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by *ov_err*.

For the definition of *ignore* see "Overflow and Integers"

3.10.59 not

arg1: EXP INTEGER(*v*)
    -> EXP INTEGER(V)

The argument is evaluated producing an integer value, of VARIETY, *v*. This value is "notted". The result is delivered as the result of the construct, with the same SHAPE as the arguments.

3.10.60 obtain_tag

t:TAG *x*
    -> EXP *x*

The value with which the TAG *t* is bound is delivered. The SHAPE of the result is the SHAPE of the value with which the TAG is bound.

3.10.61 offset_add

arg1: EXP OFFSET(*x*, *y*)
arg2: EXP OFFSET(*z*, *t*)
    -> EXP OFFSET(*x*, *t*)

The two arguments deliver OFFSETS. The result is the sum of these OFFSETS, as an OFFSET.

y will include *z*.

3.10.62 offset_div

v: VARIETY
arg1: EXP OFFSET(*x*, *x*)
arg2: EXP OFFSET(*x*, *x*)
    -> EXP INTEGER(*v*)

The two arguments deliver OFFSETS. The result is the value produced by *arg1* divided by that produced by *arg2*, as an INTEGER of VARIETY, *v*.

If the result cannot be expressed in the VARIETY being used to represent *v* the effect is undefined.

The value produced by *arg2* will be positive and non-zero.

3.10.63 offset_div_by_int

arg1: EXP OFFSET(*x*, *x*)
arg2: EXP INTEGER(*v*)
    -> EXP OFFSET(*x*, *x*)

The result is the OFFSET produced by *arg1* divided by *arg2*, as an OFFSET(*x*,*x*).

Unless the result is a signed integer multiple of the smallest possible positive OFFSET(*x*, *x*) the result is undefined.

The lower bound of *v* will be zero.

If *arg2* is zero the effect is undefined.

3.10.64 offset_max

arg1: EXP OFFSET(*x*, *y*)
arg2: EXP OFFSET(*z*, *t*)
    -> EXP OFFSET(*unite_alignments*(*x*, *z*), *intersect_alignments*(*y*, *t*))

The two arguments deliver OFFSETS. The result is the maximum of these OFFSETS, as an OFFSET.

intersect_alignments(*y*, *t*) produces an ALIGNMENT which consists of the intersection of the sets *y* and *t*. This function is not available directly as a TDF construction.

3.10.65 offset_mult

arg1: EXP OFFSET(*x*, *x*)
arg2: EXP INTEGER(*v*)
    -> EXP OFFSET(*x*, *x*)

The first argument gives an OFFSET and the second an integer. The result is the product of these, as an offset.

3.10.66 offset_negate

arg1: EXP OFFSET(*x*, *x*)
    -> EXP OFFSET(*x*, *x*)

The inverse of the argument is delivered.

3.10.67 offset_pad

a: ALIGNMENT
arg1: EXP OFFSET(*z*, *t*)
    -> EXP OFFSET(*unite_alignments*(*z*, *a*), *a*)

arg1 is evaluated. The next greater or equal OFFSET at which a value of ALIGNMENT *a* can be placed is delivered.

3.10.68 offset_pad_exp

arg1: EXP OFFSET(*a*, *b*)
arg2: EXP OFFSET(*c*, *d*)
-> EXP OFFSET(*unite_alignments*(*a*, *c*), *c*)

An obsolete construction which was C specific.

3.10.69 offset_subtract

arg1: EXP OFFSET(*x*, *y*)
arg2: EXP OFFSET(*x*, *z*)
-> EXP OFFSET(*y*, *z*)

The two arguments deliver offsets. The result is the difference of these offsets, as an offset.

Note that *x* includes *y* and *y* includes *z*, by the constraints on OFFSETS.

3.10.70 offset_test

jump_fraction: NAT_OPTION
nt: NTEST
dest: LABEL
arg1: EXP OFFSET(*x*, *y*)
arg2: EXP OFFSET(*x*, *y*)
-> EXP TOP

arg1 and *arg2* are evaluated and will produce offset values, *a* and *b*. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

a greater_than_or_equal b is equivalent to *offset_max*(*a*, *b*) = *a*, and similarly for the other comparisons.

jump_fraction is obsolete and will be ignored.

3.10.71 offset_zero

a: ALIGNMENT
-> EXP OFFSET(*a*, *a*)

A zero offset.

3.10.72 or

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(*v*)

The arguments are evaluated producing integer values of the same VARIETY, *v*. These values are "ored". The result is delivered as the result of the construct, with the same SHAPE as the arguments.

3.10.73 plus

> *ov_err*: ERROR_TREATMENT
> *arg1*: EXP INTEGER(*v*)
> *arg2*: EXP INTEGER(*v*)
> -> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The sum *a*+*b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by *ov_err*.

For the definition of *ignore* see "Overflow and Integers"

3.10.74 pointer_test

> *jump_fraction*: NAT_OPTION
> *nt*: NTEST
> *dest*: LABEL
> *arg1*: EXP POINTER(*x*)
> *arg2*: EXP POINTER(*x*)
> -> EXP TOP

arg1 and *arg2* are evaluated and will produce pointer values, *a* and *b*. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

jump_fraction is obsolete and will be ignored.

3.10.75 proc_test

> *jump_fraction*: NAT_OPTION
> *nt*: NTEST
> *dest*: LABEL
> *arg1*: EXP PROC
> *arg2*: EXP PROC
> -> EXP TOP

arg1 and *arg2* are evaluated and will produce PROC values, *a* and *b*. These values are compared using *nt*. The only permitted values of *nt* are *equal* and *not_equal*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

Two PROCS are equal if they are identical or if they were both made with *make_null_proc*. Otherwise they are unequal.

jump_fraction is obsolete and will be ignored.

3.10.76 rem1

> *arg1*: EXP INTEGER(*v*)
> *arg2*: EXP INTEGER(*v*)
> -> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* M1 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If *b* is zero the effect is undefined.

Producers may assume that suitable masking and rem1 by a power of two yield equally good code.

See the section "Division and Modulus" for the definitions of D1, D2, M1 and M2.

3.10.77 rem2

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
    -> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* M2 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If *b* is zero the effect is undefined.

Producers may assume that suitable masking and rem2 by a power of two yield equally good code if the lower bound of *v* is zero.

See the section "Division and Modulus" for the definitions of D1, D2, M1 and M2.

3.10.78 repeat

repeat_label_intro: LABEL

start: EXP TOP
body: EXP *y*
    -> EXP *y*

start is evaluated. Then *body* is evaluated.

If *body* produces a result, this is the result of the whole construction. However if *goto*(*repeat_label_intro*) or any other jump to *repeat_label_intro* is encountered during the evaluation then the current evaluation stops and *body* is evaluated again. In the canonical order all evaluated components are completely evaluated before any of the next iteration of *body*. The lifetime of *repeat_label_intro* is the evaluation of *body*.

The actual order of evaluation of the constituents shall be indistinguishable in all observable effects (apart from time) from that described above. Note that this specifically includes any defined error handling.

3.10.79 return

arg1: EXP *x*
    -> EXP BOTTOM

arg1 is evaluated to produce a value, *v*. The evaluation of the immediately enclosing procedure ceases and *v* is delivered as the result of the procedure.

Since the *return* construct can never produce a value, the SHAPE of its result is BOTTOM.

Each *apply_proc* construct specifies the SHAPE of the result it is expecting. The *return* construct actually used to provide the answer for each call will have the SHAPE specified in the corresponding *apply_proc*.

There is no other constraint on the relation between the SHAPES of *returns* used in a particular procedure.

3.10.80 round_with_mode

> *mode*: ROUNDING_MODE
> *r*: VARIETY
> *ov_err*: ERROR_TREATMENT
> *arg1*: EXP FLOATING(*f*)
> -> EXP INTEGER(*r*)

arg is evaluated to produce a floating point value, *v*. This is rounded to an integer of VARIETY, *r*, using the ROUNDING_MODE, *mode*. This is the result of the construction.

If the result cannot be expressed in the VARIETY representing R, an overflow error is caused and handled according to *ov_err*.

For the definition of *ignore* see "Overflow and Integers"

3.10.81 sequence

> *statements*: (*i*: 1, *n*)(EXP *y*[*i*])
> *result*: EXP *x*
> -> EXP *x*

The statements are evaluated in order and their results are discarded. Then *result* is evaluated and its result forms the result of the construction.

A canonical order is one in which all the components of each statement are completely evaluated before any component of the next statement is started. A similar constraint applies between the last statement and the *result*. The actual order in which the statements and their components are evaluated shall be indistinguishable in all observable effects (apart from time) from a canonical order.

Note that this specifically includes any defined error handling. However, if in a canonical order the effect of the program is undefined, the actual effect of the sequence is undefined.

Hence constructions with *impossible* error handlers may be performed before or after those with specified error handlers, if the resulting order is otherwise acceptable.

3.10.82 shape_offset

> *s*: SHAPE
> -> EXP OFFSET(*alignment*(*s*), { })

This construction delivers the minimum offset from the start of a value of SHAPE *s* to the start of any other object.

3.10.83 shift_left

> *ov_err*: ERROR_TREATMENT
> *arg1*: EXP INTEGER(*v*)
> *arg2*: EXP INTEGER(*w*)
> -> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*. The value *a* shifted left *b* places is delivered as the result of the construct, with the same SHAPE as *a*.

b will be non-negative.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by *ov_err*.

If *b* is greater than or equal to the number of bits needed to represent *v*, then the effect is undefined.

For the definition of *ignore* see "Overflow and Integers"

3.10.84 shift_right

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*w*)
-> EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*. The value *a* shifted right *b* places is delivered as the result of the construct, with the same SHAPE as the arguments.

b will be non-negative.

If the lower bound of *v* is negative the sign will be propagated.

If *b* is greater than or equal to the number of bits needed to represent *v*, then the effect is undefined.

3.10.85 subtract_ptrs

arg1: EXP POINTER(*y*)
arg2: EXP POINTER(*x*)
-> EXP OFFSET(*x*, *y*)

arg1 and *arg2* are evaluated to produce pointers *p1* and *p2*, which will be derived from the same original pointer. The result is the OFFSET from *p2* to *p1*.

3.10.86 variable

visible: BOOL
name_intro: TAG POINTER(*x*)
init: EXP *x*

body: EXP *y*
-> EXP *y*

init is evaluated to produce a value, *v*. Space is allocated to hold a value of SHAPE *x* and this is initialised with *v*. Then *body* is evaluated. During this evaluation, an original POINTER pointing to the allocated space is bound to *name_intro*. This means that inside *body* an evaluation of *obtain_tag*(*name_intro*) will produce a POINTER to this space. The lifetime of *name_intro* is the evaluation of *body*.

The value delivered by *variable* is that produced by *body*.

If *visible* is true, it means that the contents of the space may be altered while the procedure containing this declaration is not the current procedure. Hence if there are any copies of this value they will need to be refreshed from the variable when the procedure is returned to. The easiest implementation when *visible* is true may be to keep the value in memory, but this is not a necessary requirement.

The TAG given for *name_intro* will not be reused within the current UNIT. No rules for the hiding of one TAG by another are given: this will not happen.

The order in which the constituents of *init* and *body* are evaluated shall be indistinguishable in all observable effects (apart from time) from completely evaluating *init* before starting *body*. See the note about order in *sequence*.

When compiling languages which permit uninitialised variable declarations, *make_value* may be used to provide an initialisation.

3.10.87 xor

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
-> EXP INTEGER(V)

The arguments are evaluated producing integer values of the same VARIETY, *v*. These values are "xored". The result is delivered as the result of the construct, with the same SHAPE as the arguments.

3.11 EXTERNAL

An EXTERNAL defines the classes of external name available for connecting the internal names inside a CAPSULE to the world outside the CAPSULE.

3.11.1 string_extern

s: TDFIDENT(*n*)
-> EXTERNAL

string_extern produces an EXTERNAL identified by the TDFIDENT *s*.

3.11.2 unique_extern

u: UNIQUE
-> EXTERNAL

unique_extern produces an EXTERNAL identified by the UNIQUE *u*.

3.12 FLOATING_VARIETY

These describe the different kinds of floating point number which are required.

3.12.1 flvar_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> FLOATING_VARIETY

The token is applied to the arguments to give a FLOATING_VARIETY

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.12.2 flvar_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM FLOATING_VARIETY
e2: BITSTREAM FLOATING_VARIETY
-> FLOATING_VARIETY

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.12.3 flvar_parms

base: NAT
mantissa_digits: NAT
minimum_exponent: NAT
maximum_exponent: NAT
-> FLOATING_VARIETY

base is the base with respect to which the remaining numbers refer.

mantissa_digits is the required number of *base* digits, *q*, such that any number with *q* digits can be rounded to a floating point number of the variety and back again without any change to the *q* digits.

minimum_exponent is the negative of the required minimum integer such that *base* raised to that power can be represented as a non-zero floating point number in the FLOATING_VARIETY.

maximum_exponent is the required maximum integer such that *base* raised to that power can be represented in the FLOATING_VARIETY.

The *base* specified need bear no relation to the base for floating point numbers in any architecture.

A TDF translator is required to make available a representation such that, if only values within the requirements are produced, no overflow error will occur. The effect of using values outside the requirements is undefined, but an overflow may be produced.

3.13 LABEL

A LABEL marks an EXP in certain constructions, and is used in jump-like constructions to change the control to the labelled construction.

3.13.1 label_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> LABEL *x*

The token is applied to the arguments to give a LABEL.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.13.2 make_label

labelno: TDFINT
-> LABEL

Labels are represented in TDF by integers.

3.14 LINK

A LINK expresses the connection between two variables of the same SORT.

3.14.1 make_link

internal: TDFINT
ext: TDFINT
-> LINK

A LINK defines a linkable entity declared inside a UNIT as *internal* to be available to other UNITS in the same CAPSULE under the name *ext*.

LINKS are normally constructed by the TDF builder in the course of resolving sharing and name clashes when constructing a composite CAPSULE.

3.15 LINKEXTERN

A value of SORT LINKEXTERN expresses the connection between the name by which an object is known inside a CAPSULE and a name by which it is known outside.

3.15.1 make_linkextern

internal: TDFINT
ext: EXTERNAL
-> LINKEXTERN

make_linkextern produces a LINKEXTERN connecting an object identified within a CAPSULE by a TAG, TOKEN, AL_TAG or any linkable entity constructed from *internal*, with an EXTERNAL, *ext*. The EXTERNAL is an identifier which linkers and similar programs can use.

3.16 NAT

These are non-negative integers of unlimited size.

3.16.1 nat_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> NAT

The token is applied to the arguments to give a NAT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.16.2 nat_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM NAT
e2: BITSTREAM NAT
-> NAT

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.16.3 computed_nat

arg: EXP INTEGER(*var_limits*(0, *h*))
-> NAT

arg will be an install-time constant. The result is that constant.

3.16.4 make_nat

n: TDFINT
-> NAT

n is a non-negative integer of unbounded magnitude.

3.17 NTEST

These describe the comparisons which are possible in the various *test* constructions. Note that *greater_than* is not necessarily the same as *not_less_than_or_equal*, since the result need not be defined (eg. in IEEE floating point).

3.17.1 ntest_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> NTEST

The token is applied to the arguments to give a NTEST.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.17.2 ntest_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM NTEST

e2: BITSTREAM NTEST
-> NTEST

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.17.3 equal

-> NTEST

Signifies "equal" test.

3.17.4 greater_than

-> NTEST

Signifies "greater than" test.

3.17.5 greater_than_or_equal

-> NTEST

Signifies "greater than or equal" test.

3.17.6 less_than

-> NTEST

Signifies "less than" test.

3.17.7 less_than_or_equal

-> NTEST

Signifies "less than or equal" test.

3.17.8 not_equal

-> NTEST

Signifies "not equal" test.

3.17.9 not_greater_than

-> NTEST

Signifies "not greater than" test.

3.17.10 not_greater_than_or_equal

-> NTEST

Signifies "not (greater than or equal)" test.

3.17.11 not_less_than

-> NTEST

Signifies "not less than" test.

3.17.12 not_less_than_or_equal

-> NTEST

Signifies "not (less than or equal)" test.

3.18 PROPS

A PROPS is an assemblage of program information. This standard offers five ways of constructing a PROPS - ie. it defines five kinds of information which it is useful to express. These are:

definitions of AL_TAGs standing for ALIGNMENTs;

declarations of TAGs standing for EXPs;

definitions of the EXPs for which TAGs stand;

declarations of TOKENs standing for pieces of program;

definitions of the pieces of program for which TOKENs stand.

The standard can be extended by the definition of new kinds of PROPS information and new PROPS constructs for expressing them; and private standards can define new kinds of information and corresponding constructs without disruption to adherents to the present standard.

3.18.1 make_al_tagdefs

no_labels: TDFINT
tds: (*i*: 1, *n*) AL_TAGDEF[*i*]
-> AL_TAGDEF_PROPS

no_labels is the number of local LABELS used in *tds*. *tds* is a list of AL_TAGDEFS which define the bindings for *al_tags*.

A PROPS is produced, suitable for incorporation into a UNIT.

3.18.2 make_tagdecs

> *no_labels*: TDFINT
> *tds*: (*i*: 1, *n*) TAGDEC[*i*]
> -> TAGDEC_PROPS

no_labels is the number of local LABELS used in *tds*. *tds* is a list of TAGDECS which declare the SHAPES associated with TAGS.

A PROPS is produced, suitable for incorporation into a UNIT.

3.18.3 make_tagdefs

> *no_labels*: TDFINT
> *tds*: (*i*: 1, *n*) TAGDEF[*i*]
> -> TAGDEF_PROPS

no_labels is the number of local LABELS used in *tds*. *tds* is a list of TAGDEFS which give the EXPS which are the definitions of values associated with TAGS.

A PROPS is produced, suitable for incorporation into a UNIT.

3.18.4 make_tokdecs

> *tds*: (*i*: 1, *n*) TOKDEC[*i*]
> -> TOKDEC_PROPS

tds is a list of TOKDECS which gives the sorts associated with TOKENS.

A PROPS is produced, suitable for incorporation into a UNIT.

3.18.5 make_tokdefs

> *no_labels*: TDFINT
> *tds*: (*i*: 1, *n*) TOKDEF[*i*]
> -> TOKDEF_PROPS

no_labels is the number of local LABELS used in *tds*. *tds* is a list of TOKDEFS which gives the definitions associated with TOKENS.

A PROPS is produced, suitable for incorporation into a UNIT.

3.19 ROUNDING_MODE

ROUNDING_MODE specifies the way rounding is to be performed in floating point arithmetic.

3.19.1 rounding_mode_apply_token

> *token_value*: TOKEN
> *token_args*: BITSTREAM

-> ROUNDING_MODE

The token is applied to the arguments to give a ROUNDING_MODE.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.19.2 rounding_mode_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM ROUNDING_MODE
e2: BITSTREAM ROUNDING_MODE
-> ROUNDING_MODE

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.19.3 round_as_state

-> ROUNDING_MODE

Round as specified by the current state of the machine.

3.19.4 to_nearest

-> ROUNDING_MODE

Signifies rounding to nearest. The effect when the number lies half-way is not specified.

3.19.5 toward_larger

-> ROUNDING_MODE

Signifies rounding toward next largest.

3.19.6 toward_smaller

-> ROUNDING_MODE

Signifies rounding toward next smallest.

3.19.7 toward_zero

-> ROUNDING_MODE

Signifies rounding toward zero.

3.20 SHAPE

SHAPES express symbolic size and representation information about run time values.

SHAPES are constructed from primitive SHAPES which describe values such as procedures and integers, and recursively from compound construction in terms of other SHAPES.

3.20.1 shape_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> SHAPE

The token is applied to the arguments to give a SHAPE.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.20.2 shape_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM SHAPE
e2: BITSTREAM SHAPE
-> SHAPE

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.20.3 bitfield

bf_var: BITFIELD_VARIETY
-> SHAPE

A BITFIELD is used to represent a pattern of bits which may be packed. Installers shall represent these bits compactly; there shall be no intervening bits.

A BITFIELD_VARIETY specifies the number of bits and whether they are considered to be signed. If an bitfield_variety has *n* bits, the numbers (0..2^*n*-1) shall be represented if the variety is unsigned, and (-2^*n*..2^*n*-1) if signed.

There are very few operations on BITFIELDS, which have to be converted to INTEGERS before arithmetic can be performed on them.

An installer may place a limit on the number of bits it implements. Such a limit shall not be less than 32 bits. A statement of any such limits shall be part of the specification of an installer. Installers are encouraged not to place any such limit on BITFIELDS.

3.20.4 bottom

-> SHAPE

BOTTOM is the SHAPE which describes a piece of program which does not evaluate to any result. Examples include *goto* and *return*.

If BOTTOM is a parameter to any other SHAPE constructor, the result is BOTTOM.

3.20.5 compound

sz: EXP OFFSET(*x*, { })
-> SHAPE

The SHAPE constructor COMPOUND describes cartesian products and unions.

sz will evaluate to a constant. The resulting SHAPE describes a value whose size is given by *sz*.

3.20.6 floating

fv: FLOATING_VARIETY
-> SHAPE

Most of the floating point arithmetic operations, *floating_plus*, *floating_minus* etc., are defined to work in the same way on different kinds of floating point number. If these operations have more than one argument the arguments have to be of the same kind, and the result is of the same kind.

An installer may limit the FLOATING_VARIETIES it can represent. A statement of any such limits shall be part of the specification of an installer.

3.20.7 integer

var: VARIETY
-> SHAPE

The different kinds of INTEGER are distinguished by having different VARIETIES. A fundamental VARIETY (not a TOKEN or conditional) is represented by two SIGNED_NATS, respectively the lower and upper bounds (inclusive) of the set of values belonging to the VARIETY. Implementers will represent such a variety by an object of sufficient size or more.

Most architectures require that dyadic integer arithmetic operations take arguments of the same size, and so TDF does likewise. Because TDF is completely architecture neutral and makes no assumptions about word length, this means that the VARIETIES of the two arguments must be identical. An example illustrates this. A piece of TDF which attempted to add two values whose SHAPES were

INTEGER(0,60000) and INTEGER(0, 30000)

would be undefined. The reason is that without knowledge of the target architecture's word length, it is impossible to guarantee that the two values are going to be represented in the same number of bytes. On a 16-bit machine they probably would, but not on a 15-bit machine. The only way to ensure that two INTEGERS are going to be represented in the same way in all machines is to stipulate that their VARIETIES are exactly the same.

When any construct delivering an INTEGER of a given VARIETY produces a result which is not representable in the space which an installer has chosen to represent that VARIETY, an integer overflow occurs. Whether it occurs in a particular case depends on the target, because the installers' decisions on representation are inherently target-defined.

A particular installer may limit the ranges of integers that it implements. Every installer shall implement all unsigned ranges included in INTEGER(0,2ˆ32-1), and all signed ranges included in INTEGER(-2ˆ31,2ˆ31-1). A statement of any such limits shall be part of the specification of an installer. Installers are encouraged not to place any upper bound on the ranges of integers.

3.20.8 local_label_value

-> SHAPE

LOCAL_LABEL_VALUE is the SHAPE which describes a local label value, for creation by *make_local_lv* and use by *goto_local_lv*.

3.20.9 nof

n: NAT
s: SHAPE
-> SHAPE

The NOF constructor describes the SHAPE of a value consisting of an array of *n* values of the same SHAPE, *s*. The SHAPE, *s*, may be TOP and *n* may be zero.

3.20.10 offset

arg1: ALIGNMENT
arg2: ALIGNMENT
-> SHAPE

The SHAPE constructor OFFSET describes values which represent the differences between POINTERS, that is they measure offsets in memory. It should be emphasised that these are in general run-time values.

An OFFSET measures the displacement from the value indicated by a POINTER(*arg1*) to the value indicated by a POINTER(*arg2*). Such an offset is only defined if the POINTERS are derived from the same original POINTER.

The set *arg1* will include the set *arg2*.

3.20.11 pointer

arg: ALIGNMENT
-> SHAPE

A POINTER is a value which points to space allocated in a computer's memory. The POINTER constructor takes an ALIGNMENT argument.

3.20.12 proc

-> SHAPE

PROC is the SHAPE which describes pieces of program which deliver procedure values.

3.20.13 top

-> SHAPE

TOP is the SHAPE which describes pieces of program which return no useful value. *assign* is an example: it performs an assignment, but does not deliver any useful value.

3.21 SIGNED_NAT

These are positive or negative integers of unbounded size.

3.21.1 signed_nat_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> SIGNED_NAT

The token is applied to the arguments to give a SIGNED_NAT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.21.2 signed_nat_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM SIGNED_NAT
e2: BITSTREAM SIGNED_NAT
-> SIGNED_NAT

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.21.3 computed_signed_nat

arg: EXP INTEGER(*v*)
-> SIGNED_NAT

arg will be an install-time constant. The result is that constant.

3.21.4 make_signed_nat

neg: TDFBOOL
n: TDFINT
-> SIGNED_NAT

n is a non-negative integer of unbounded magnitude. The result is negative iff *neg* is true.

3.21.5 snat_from_nat

neg: BOOL
n: NAT
-> SIGNED_NAT

The result is negative iff *neg* is true.

3.22 SORTNAME

These are the name of the SORTS which can be parameters of TOKEN definitions.

3.22.1 al_tag

-> SORTNAME

3.22.2 alignment_sort

-> SORTNAME

3.22.3 bitfield_variety

-> SORTNAME

3.22.4 bool

-> SORTNAME

3.22.5 error_treatment

-> SORTNAME

3.22.6 exp

-> SORTNAME

3.22.7 floating_variety

-> SORTNAME

3.22.8 foreign_sort

foreign_name: TDFSTRING
-> SORTNAME

This SORT enables unanticipated kinds of information to be placed in TDF.

3.22.9 label

-> SORTNAME

3.22.10 nat

-> SORTNAME

3.22.11 ntest

-> SORTNAME

3.22.12 rounding_mode

-> SORTNAME

3.22.13 shape

-> SORTNAME

3.22.14 signed_nat

-> SORTNAME

3.22.15 tag

-> SORTNAME

3.22.16 token

params: $(i\colon 1, n)(s[i]$: SORTNAME)
result: SORTNAME
-> SORTNAME

The SORTNAME of a TOKEN. Note that it can have tokens as parameters.

3.22.17 variety

-> SORTNAME

3.23 TAG

These are used to name values and variables in the run time program.

3.23.1 tag_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> TAG *x*

The token is applied to the arguments to give a TAG.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.23.2 make_tag

tagno: TDFINT
-> TAG *x*

make_tag produces a TAG identified by *tagno*.

3.24 TAGDEC

A TAGDEC declares a TAG for incorporation into a UNIT.

3.24.1 make_id_tagdec

t_intro: TDFINT
x: SHAPE
-> TAGDEC

A TAGDEC announcing that the TAG *t_intro* identifies an EXP of SHAPE *x* is constructed.

3.24.2 make_var_tagdec

t_intro: TDFINT
x: SHAPE
-> TAGDEC

A TAGDEC announcing that the TAG *t_intro* identifies an EXP of SHAPE POINTER(*alignment*(*x*)) is constructed.

3.25 TAGDEF

A value of SORT TAGDEF gives the definition of a TAG for incorporation into a UNIT.

3.25.1 make_id_tagdef

t: TDFINT

e: EXP *x*
-> TAGDEF

make_id_tagdef produces a TAGDEF defining the TAG constructed from the TDFINT, *t*. This TAG is defined to stand for the value delivered by *e*.

e will be a constant evaluable at load_time.

3.25.2 make_var_tagdef

t: TDFINT

e: EXP *x*
  -> TAGDEF

make_var_tagdef produces a TAGDEF defining the TAG constructed from the TDFINT, *t*. This TAG stands for a variable which is initialised with the value delivered by *e*.

e will be a constant evaluable at load_time.

3.26 TDFBOOL

A TDFBOOL is the TDF encoding of a boolean.

3.27 TDFIDENT

A TDFIDENT(*n*) encodes a sequence of *n* unsigned integers of size 8 bits. It is byte aligned. This construction will not be used inside a BITSTREAM.

3.28 TDFINT

A TDFINT is the TDF encoding of an unbounded unsigned integer constant.

3.29 TDFSTRING

A TDFSTRING(*k*, *n*) encodes a sequence of *n* unsigned integers of size *k* bits.

3.30 TOKDEC

A TOKDEC declares a TOKEN for incorporation into a UNIT.

3.30.1 make_tokdec

tok: TDFINT
s: SORTNAME
  -> TOKDEC

The sort of the token *tok* is declared to be *s*.

3.31 TOKDEF

A TOKDEF gives the definition of a TOKEN for incorporation into a UNIT.

3.31.1 make_tokdef

tok: TDFINT
def: BITSTREAM(*result_sort*: SORTNAME, *params*: (*i*: 1, *n*)(*sn*[*i*]: SORTNAME, *tk*[*i*]: TDFINT), *body*: *result_sort*)
-> TOKDEF

A TOKDEF is constructed which defines the TOKEN *tok* to stand for the fragment of TDF, *body*, which may be of any SORT with a SORTNAME, except for *token*.

At the application of this TOKEN actual pieces of TDF having SORT *sn*[*i*] are supplied to correspond to the *tk*[*i*]. The application denotes the piece of TDF obtained by substituting these actual parameters for the corresponding TOKENS within *body*.

The body need not be re-evaluated if there are no parameters.

TOKEN definitions will not be recursive, that is, they will not be applied in *body*, or in any *token* application invoked in applying the *body*.

3.32 TOKEN

These are used to functions evaluated at installation time. They are represented by TDFINTS.

3.32.1 token_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> TOKEN

The token is applied to the arguments to give a TOKEN.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.32.2 make_tok

tokno: TDFINT
-> TOKEN

make_tok constructs a TOKEN identified by *tokno*.

3.33 UNIQUE

These are used to provide world-wide unique names for TOKENS.

3.33.1 make_unique

text: (*i*: 1, *n*) (*s*[*i*] : TDFIDENT(*m*[*i*]))

-> UNIQUE

Two UNIQUE values are equal iff they were constructed with equal arguments.

3.34 UNIT

A UNIT gathers together a PROPs and LINKs which relate the names by which objects are known inside the PROPS and names by which they are to be known across the whole of the enclosing CAPSULE.

3.34.1 make_unit

local_vars: (*i*: 1, *nlv*) (*nl*[*i*] : TDFINT)

lks: (*r*: 1, *nlv*) ((*s*[*r*]: 1, *n*[*r*]) *lk*[*s*[*r*]] : LINK)
properties: BYTESTREAM
-> UNIT

local_vars gives the the number of linkable entities of each kind. These numbers correspond (in the same order) to the variable sorts in *capsule_linking* in *make_capsule*. The linkable entities will be represented by TDFINTS in the range 0 to *nl*[*i*].

lks gives the LINKS for each kind of entity in the same order as in *local_vars*.

The *properties* will be of a form dictated by the kind of UNIT, this may be deduced from the information as specified in *make_capsule*.

nlv will be either 0 or equal to the value of *n_link* used in *make_capsule*.

3.35 VARIETY

These describe the different kinds of integer which can occur at run time. The fundamental construction consists of a SIGNED_NAT for the lower bound of the range of possible values, and a SIGNED_NAT for the upper bound (inclusive at both ends).

The existence of the constructions *and*, *not*, *or* and *xor* implies that the representation of integers is binary.

There is no limitation on the magnitude of these bounds in TDF, but an installer may specify limits. Every installer shall implement all unsigned ranges included in INTEGER(0,2^32-1), and all signed ranges included in INTEGER(-2^31,2^31-1). A statement of any such limits shall be part of the specification of an installer. Installer are encouraged not to place any upper bound on the ranges of integers.

3.35.1 var_apply_token

token_value: TOKEN
token_args: BITSTREAM
-> VARIETY

The token is applied to the arguments to give a VARIETY.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTS of its parameters, in the order specified.

3.35.2 var_cond

control: EXP INTEGER(*v*)
e1: BITSTREAM VARIETY
e2: BITSTREAM VARIETY
-> VARIETY

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

3.35.3 var_limits

lower_bound: SIGNED_NAT
upper_bound: SIGNED_NAT
-> VARIETY

lower_bound is the lower limit (inclusive) of the range of values which shall be representable in the resulting VARIETY, and *upper_bound* is the upper limit (inclusive).

4. Notes

4.1 Binding

The following constructions introduce TAGS:- *identify*, *variable*, *make_proc*, *make_id_tagdec*, *make_var_tagdec*.

During the evaluation of *identify* and *variable* a value, *v*, is produced which is bound to the TAG during the evaluation of an EXP or EXPS. The TAG is "in scope" for these EXPS. This means that in the EXP a use of the TAG is permissible and will refer to the declaration.

The *make_proc* construction introduces TAGS which are bound to the actual parameters on each call of the procedure. These TAGS are "in scope" for the body of the procedure.

The *make_id_tagdec* and *make_var_tagdec* constructions introduce TAGS which are "in scope" throughout the *tagdef* UNITS. These TAGS may have values defined for them in the *tagdef* UNITS, or values may be supplied by linking.

The following constructions introduce LABELS:- *conditional*, *repeat*, *labelled*.

The construction themselves define EXPS for which these LABELS are "in scope". This means that in the EXPS a use of the LABEL is permissible and will refer to the introducing construction.

TAGS and LABELS introduced in the body of a TOKEN definition are systematically renamed in their scope, which will be completely included by the body, each time the TOKEN definition is applied.

Each of the values introduced in a UNIT will be named by a different TAG, and the labelling constructions will use different labels, so no visibility rules are needed. The set of TAGS and LABELS used in a simple UNIT are considered separately from those in another simple UNIT, so no question of visibility arises. The compound and link UNITS provide a method of relating the items in one simple UNIT to those in another, but this is through the intermediary of yet another set of TAGS and TOKENS.

4.2 Character codes

TDF does not have a concept of characters. It transmits integers of various sizes. So if a producer wishes to communicate characters to an installer, it will usually have to do so by encoding them in some way as integers.

An ANSI C producer sending a TDF program to a set of normal C environments may well choose to encode its characters using the ASCII codes, an EBCDIC based producer transmitting to a known set of EBCDIC environments might use the code directly, and a wide character producer might likewise choose a specific encoding. For some programs this way of proceeding is necessary, because the codes are used both to represent characters and for arithmetic, so the particular encoding is enforced. In these cases it will not be possible to translate the characters to another encoding because the character codes will be used in the TDF as ordinary integers, which must not be translated.

Some producers may wish to transmit true characters, in the sense that something is needed to represent particular printing shapes and nothing else. These representations will have to be transformed into the correct character encoding on the target machine.

Probably the best way to do this is to use TOKENS. A fixed representation for the printing marks could be chosen in terms of integers and TOKENS introduced to represent the translation from these integers to local character codes, and from strings of integers to strings of local character codes. These definitions could be bound on the target machine and the installer should be capable of translating these constructions into efficient machine code. To make this a standard, unique TOKENS should be used.

But this raises the question, who chooses the fixed representation and the unique TOKENS and their specification? Clearly TDF provides a mechanism for performing the standardisation without itself defining a standard.

Here TDF gives rise to the need for extra standards, especially in the specification of globally named unique TOKENS.

4.3 Constant evaluation

Some constructions require an EXP argument which is "constant at install time". For an EXP to satisfy this condition it must be constructed according to the following rules after substitution of token definitions and selection of *exp_cond* branches.

If it contains *obtain_tag* then the tag will either be introduced within the EXP, or it will be introduced with *make_var_tagdef* and neither *contents* nor *assign* will be applied to it nor to any POINTER derived from it.

It may not contain any of the following constructions:- *assign_to_volatile*, *contents_of_volatile*, *current_env*, *goto_local_lv*, *make_local_lv*, *move_some*, *repeat*, *round_as_state*.

It may not contain *labelled* if there are jumps between any of the branches.

4.4 Division and modulus

Two classes of division(D) and remainder(M) construct are defined. The two classes have the same definition if both operands have the same sign. Neither is defined if the second argument is zero.

Class 1:

$p \text{ D1 } q = n$

where

$p = n*q + (p \text{ M1 } q)$

$\text{sign}(p \text{ M1 } q) = \text{sign}(q)$

$0 <= |p \text{ M1 } q| < |q|$

Class 2:

$p \text{ D2 } q = n$

where

$p = n*q + (p \text{ M2 } q)$

sign(p M2 q) = sign(p)

0 <= $|p$ M2 $q| < |q|$

4.5 Encoding primitive values

The primitives in the encoding are:-

TDFBOOL, which encodes 0 and 1.

TDFINT, which encodes non-negative integers of unbounded size.

TDFSTRING(k, n), which encodes a sequence of n integers of size k bits.

TDFIDENT(n), which encodes a sequence of n integers of size 8 bits. The encoding is different from that of TDFSTRING, in that it is byte-aligned.

BITSTREAM, which encodes a measure of length followed by an item of that length. This permits the following item to be skipped. The following item will usually be an encoding of some TDF entity. If so, and if its *sort* is known, then this is indicated.

BYTESTREAM, which encodes a measure of length followed by an item of that length. This permits the following item to be skipped. The following item will usually be an encoding of some TDF entity. If so, and if its *sort* is known, then this is indicated. This kind of item is byte-aligned.

4.6 Equality of EXPS

A definition of equality of EXPS would be a considerable part of a formal specification of TDF, and is not given here.

4.7 Equality of SHAPES

Equality of SHAPES is defined recursively.

Two SHAPES are equal if they are both BOTTOM, or both TOP or both PROC or both LOCAL_LABEL_VALUE.

Two SHAPES are equal if they are both *integer*, both *floating*, or both *bitfield*, and the corresponding parameters are equal.

Two SHAPES are equal if they are both NOF, the numbers of items are equal and the SHAPE parameters are equal.

Two OFFSETS or two POINTERS are equal if their ALIGNMENT parameters are pairwise equal.

Two COMPOUNDS are equal if their offset expressions are equal.

No other pairs of SHAPES are equal.

4.8 Exceptions and jumps

TDF allows simply for labels and jumps within a procedure, by means of the *conditional*, *labelled* and *repeat* constructions, and the *goto*, *case* and various *test* constructions. But there are two more complex jumping situations.

First there is the jump, known to stay within a procedure, but to a computed destination. Many languages have discouraged this kind of construction, but it is still available in Cobol (implicitly), and it can be used to provide other facilities (see below). TDF allows it by means of the LOCAL_LABEL_VALUE. TDF is arranged so that this can usually be implemented as the address of the label. The *goto_local_lv* construction just jumps to the label. The setting of the stack in these circumstances can be optimised.

The other kind of construction needed is the jump out of a procedure to a label which is still active, restoring the environment of the destination procedure: the long jump. Related to this is the notion of exception. Unfortunately long jumps and exceptions do not co-exist well. Exceptions are commonly organised so that any necessary destruction operations are performed as the stack of frames is traversed; long jumps commonly go directly to the destination. TDF must provide some facility which can express both of these concepts. Furthermore exceptions come in several different versions, according to how the exception handlers are discriminated and whether exception handling is initiated if there is no handler which will catch the exception.

Fortunately the normal implementations of these concepts provide a suggestion as to how they can be introduced into TDF. The local label value provides the destination address, the environment (produced by *current_env*) provides the stack frame for the destination, and the stack re-setting needed by the local label jumps themselves provides the necessary stack information. If more information is needed, such as which exception handlers are active, this can be created by producing the appropriate TDF.

So TDF takes the long jump as the basic construction, and its parameters are a local label value and an environment. Everything else can be built in terms of these.

4.9 Frames

TDF states that while a particular procedure activation is current, it is possible to create a POINTER, by using *current_env*, which gives access to all the declared variables and identifications of the activation which are alive and which have been marked as visible. The construction *env_offset* gives the OFFSET of one of these relative to such a POINTER. These constructions may serve for several purposes.

One significant purpose is to implement such languages as Pascal which have procedures declared inside other procedures. One way of implementing this is by means of a "display", that is, a tuple of frame pointers of active procedures.

Another purpose is to find active variables satisfying some criterion in all the procedure activations. This is commonly required for garbage collection. TDF does not force the installer to implement a frame pointer register, since some machines do not work best in this way. Instead, a frame pointer is created only if required by *current_env*. The implication of this is that this sort of garbage collection needs the collaboration of the producer to create TDF which makes the correct calls on *current_env* and *env_offset* and place suitable values in known positions.

Programs compiled especially to provide good diagnostic information can also use these operations.

In general any program which wishes to manipulate the frames of procedures other than the current one can use *current_env* and *env_offset* to do so.

4.10 Recommendations about VARIETIES and FLOATING_VARIETIES

VARIETIES and FLOATING_VARIETIES should reflect as accurately as possible what is needed by the program. This choice should not be influenced by the knowledge of what is available on common machines (except where the purpose is specifically to take advantage of this knowledge). It is the task of the TDF translator to make intelligent decisions.

4.11 Lifetimes

TAGS are bound to values during the evaluation of EXPS, which are specified by the construction which introduces the TAG. The evaluation of these EXPS is called the lifetime of the activation of the TAG.

Note that lifetime is a different concept from that of scope. For example, if the EXP contains the application of a procedure, the evaluation of the body of the procedure is within the lifetime of the TAG, but the TAG may well not be in scope.

A similar concept applies to LABELS.

4.12 Memory Model

The layout of data in memory is entirely determined by the calculation of OFFSETS relative to POINTERS. That is, it is determined by OFFSET arithmetic and the *add_to_ptr* construction.

A POINTER is parameterised by the ALIGNMENT of the data indicated. An ALIGNMENT is a set of all the different kinds of basic value which can be indicated by a POINTER. That is, it is a set chosen from all VARIETIES, all BITFIELD_VARIETIES, all FLOATING_VARIETIES, *proc*, *local_label_value*, *pointer* and *offset*.

The implication of this is that the ALIGNMENT of all procedures is the same, the ALIGNMENT of all LOCAL_LABEL_VALUES is the same, the ALIGNMENT of all POINTERS is the same and the ALIGNMENT of all offsets is the same.

At present this corresponds to the state of affairs for all machines. But it is certainly possible that, for example, 64-bit pointers might be aligned on 64-bit boundaries while 32-bit pointers are aligned on 32-bit boundaries. In this case it will become necessary to add different kinds of pointer. This will not present a problem should TDF have to change, because, to use such pointers, similar changes will have to be made in languages to distinguish the kinds of pointer if they are to be mixed.

The difference between two POINTERS is measured by an OFFSET. Hence an OFFSET is parameterised by two ALIGNMENTS, that of the starting POINTER and that of the end POINTER. The ALIGNMENT set of the first must include or be equal to the ALIGNMENT set of the second.

The operations on OFFSETS are subject to various constraints on ALIGNMENTS. It is important not to read into offset arithmetic what is not there. Accordingly a very rough guide to the algebra of OFFSETS is given below.

offset_add is associative.

offset_zero is a zero for *offset_add* where it is a legal parameter.

offset_pad(offset_zero(a), b) is a zero for *offset_add* where it is a legal parameter.

offset_subtract and *offset_negate* obey the conventional laws where they form legal expressions. For example,

$a + b - b = a$ etc.

offset_mult corresponds to repeated offset_addition.

offset_max is commutative, associative and idempotent.

offset_add distributes over *offset_max* where they form legal expressions.

offset_test(>= , *a*, *b*) continues if *offset_max*(*a*, *b*) = *a*, etc.

An example of the representation of OFFSET arithmetic is given below. This is not a definition, but only an example. In order to make this clear a machine with bit addressing is hypothesized.

In this machine ALIGNMENTS will be represented by the number by which the bit address of data must be divisible. For example, 8-bit bytes will have an ALIGNMENT of 8, longs of 32 and doubles of 64. OFFSETS will be represented by the displacement in bits from a POINTER. POINTERS will be represented by the bit address of the data. Only one memory space will exist. Then in this example a possible conforming implementation would be as follows.

add_to_ptr is addition.

offset_add is addition.

offset_div and *offset_div_by_int* are exact division.

offset_max is maximum.

offset_mult is multiply.

offset_negate is negate.

offset_pad(*a*, *x*) is $((x + a - 1) / a) * a$

offset_subtract is subtract.

offset_test is *integer_test*.

offset_zero is 0.

shape_offset(*s*) is the minimum number of bits needed to be moved to move a value of SHAPE *s*.

Note that these operations only exist where the constraints on the parameters are satisfied. Elsewhere the operations are undefined.

4.13 Order of evaluation

The order of evaluation is specified in certain constructions in terms of equivalent effect with a canonical order of evaluation. These constructions are *conditional*, *identify*, *labelled*, *repeat*, *sequence* and *variable*. Let these be called the order-specifying constructions.

The constructions which change control also specify a canonical order. These are *apply_proc*, *case*, *goto*, *goto_local_lv*, *long_jump*, *return*, the *test* constructions and all instructions containing the *error_jump* ERROR_TREATMENT. In the canonical evaluation of these constructions the evaluation of the new EXP starts after the construction is obeyed.

The construction *apply_proc* specifies the order in respect of the procedure evaluation but the order of evaluation of parameters is specified below.

The order of evaluation of the components of other constructions and of the parameters of *apply_proc* is as follows. The components may be evaluated in any order and with their components - down to the TDF leaf level - interleaved in any order. The constituents of the order specifying constructions may also be interleaved in any order, but the order of the operations within an order specifying operation shall be equivalent in effect to a canonical order. All the components are evaluated before the operation of the construction is itself applied.

4.14 Original pointers

Certain constructions are specified as producing original pointers. They allocate space to hold values and produce pointers indicating that new space. All other pointer values are derived pointers, which are produced from original pointers by a sequence of *add_to_ptr* operations. Counting original pointers as being derived from themselves, every pointer is derived from just one original pointer.

4.15 Overflow and Integers

It is necessary first to define what overflow means for integer operations and second to specify what happens when it occurs. The intention of TDF is to permit the simplest possible implementation of common constructions on all common machines while allowing precise effects to be achieved, if necessary at extra cost.

Integer varieties may be represented in the computer by any range of integers which includes the bounds given for the variety. An arithmetic operation may therefore yield a result which is within the stated variety, or outside the stated variety but inside the range of representing values, or outside that range. Most machines provide instructions to detect the latter case, but testing for the second case is possible but a little more costly.

In the first two cases the result is defined to be the value in the representation. Overflow occurs only in the third case.

If the ERROR_TREATMENT is *impossible* overflow will not occur. If it should happen to do so the effect of the operation is undefined.

If the ERROR_TREATMENT is *error_jump* a LABEL is provided to jump to if overflow occurs.

The *ignore* ERROR_TREATMENT is provided so that a useful defined result may be produced in certain cases where it is usually easily available on most machines. This result is available on the assumption that machines use binary arithmetic for integers. This is certainly so at present, and there is no close prospect of other bases being used. Hence if the lower bound of the variety is zero and the upper bound is 2ˆn and the result in unbounded arithmetic is r, the result in case of overflow is defined to be a number which is congruent to r modulo 2ˆn and lies in the representation. In all other cases the effect is undefined.

If a precise result is required further arithmetic and testing may be needed which the installer may be able to optimise away if the word lengths happen to suit the problem. In extreme cases it may be necessary to use a larger variety.

4.16 Reference Model

If a TDF CAPSULE contains more than one definition or declaration of an entity the question arises as to whether this is legal TDF and if so what it means.

Each GROUP of UNITS is considered to contain a collection of statements linked by a commutative, associative and idempotent operation. That is, the order of statements within a GROUP is immaterial. This operation is written as "^". For the tokdec, tokdef, al_tagdef, tagdec and tagdef GROUPS the understanding of multiple declarations or definitions is

dec_or_def(*a*, *d1*) ^ dec_or_def(*a*, *d2*)

is equivalent to

dec_or_def(*a*, *d1* | *d2*)

where "|" is an associative, commutative and idempotent operation which is particular to the kind of GROUP.

For tokdec, tokdef, al_tagdef and tagdec GROUPS *d1* and *d2* will be equal.

For the tagdef GROUP *d1* and *d2* will satisfy one of the following criteria.

1) *d1* is equal to *d2*.

2) *d1* and *d2* are *nof*(*n*, *s*) and *nof*(*m*, *s*) respectively, and either $n<m$ and the first *n* components of *d2* are equal to the components of *d1* in the same order, or else $n>m$ and the first *m* components of *d1* are equal to the components of *d2* in the same order.

3) *d1* and *d2* are of the form *make_compound*(*q1*) and *make_compound*(*q2*) respectively. Considering *q1* and *q2* as sets of pairs of OFFSETS and values, either *q1* is included in *q2* or *q2* is included in *q1*.

4.17 SHAPES: Least Upper Bound

The LUB of two SHAPES, *a* and *b* is defined as follows.

If *a* and *b* are equal shapes, then *a*.

If *a* is BOTTOM then *b*.

If *b* is BOTTOM then *a*.

Otherwise TOP.

Since this definition is commutative, associative and idempotent it can be extended conventionally to sets of SHAPES.

4.18 Standard tokens

There are many examples where it would be desirable to have standard tokens defined, which are such that any producer can use them and any installer with the necessary properties will implement them.

The TDF principle is that every construction in TDF is implemented in every installer. Any facilities which are required in some installers but which cannot be usefully implemented on all installers should be provided by means of tokens. These tokens might be substituted in the usual way, or they might be specifically recognised by the installers. IEEE floating point is an example. It would not be desirable to specify floating point to meet the IEEE standard, because there are important machines which do not implement it. But it should be possible to access the operations on the right kind of machine. A set of tokens can be defined to make this possible. But it is clearly desirable that all producers and installers using IEEE should use the same tokens, so a standard set of tokens and their meaning is needed.

Another kind of example is the tokens which are used to shorten the transmitted TDF, and are substituted during installation. Tokens for the booleans, true and false, would be an example. Again it is a good idea to have standards for these purposes.

Such standards should be kept out of the TDF specification, since they can exist independently. But it is important that they should exist. DRA will be working with OSF and USL to create such standards.

4.19 TDF is not closed

TDF is not, and is not intended to be, semantically closed or complete. TDF is designed to permit the distribution of incomplete parts of programs. Such programs can be linked on the target machine with modules which can produce values which will be used by the TDF parts of the complete program.

An important example of this, when TDF is used for C programs, is the use of malloc, calloc and free. These produce and use pointers which have a semantics (particularly a lifetime) which they define. The semantics of the whole program has to take into account the semantics of the parts which did not arise from TDF as well as the TDF semantics.

4.20 Token Libraries

It is intended that libraries of tokens should be defined to assist in the implementation of languages. For example, DRA will provide such a library for ANSI C. It is very desirable that these libraries should be standardised, as otherwise they will be re-invented, probably incompatibly. They are not part of the definition of TDF, but they are very closely associated with it.

CONTENTS