UNLIMITED

(2)

Report No. 91005

Report No. 91005

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

TDF SPECIFICATION

Authors: J M Foster, M Brandreth, P W Core, I F Currie & N E Peeling

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

October 1990

UNLIMITED

CONDITIONS OF RELEASE

0100256 301954

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 91005

Title: TDF Specification

Authors: J M Foster
M Brandreth
P W Core
I F Currie
N E Peeling

Date: October 1990

Summary

This report is primarily intended to serve as a reference manual which precisely defines the meaning of TDF. An extended introduction is included which explains the purpose of TDF and gives an overview of its structure. It also explains those concepts within the definition which may be new to the reader and defines terminology used in the rest of the document.

Some discussion sections are also included amongst the definitions of TDF constructs. These can be read independently of the definitions and are intended to provide the reader with a broader perspective of particular areas of TDF.

1 Introduction

1.1 TDF: Scenario of Use

1.2 TDF: Level of Definition

1.3 Values within a TDF System

1.3.1 Dynamic Values
1.3.2 Static Values
1.3.3 SORTs and SHAPEs: an Example
1.3.4 SHAPE- and SORT-correctness

1.4 Identification of Values

1.5 TDF: Architecture Neutrality

1.5.1 SHAPEs SIZEs and OFFSETs
1.5.1.1 TUPLE
1.5.1.2 UNION
1.5.1.3 Arrays: NOF and SOME
1.5.1.3.1 NOF
1.5.1.3.2 SOME
1.5.1.4 SIZE and OFFSET
1.5.1.4.1 SIZE
1.5.1.4.2 OFFSET
1.5.1.5 SHAPEs SIZEs and OFFSETs: ANSI C
1.5.2 Conditional Compilation
1.5.3 Tokenisation

1.6 Structure of a TDF Capsule

1.7 TDF Terminology

1.7.1 Specifying Translator Behaviour
1.7.2 Describing Program Construction

2 Definition

2.1 TDF Level 0

2.1.1 Level 0 SORTs
2.1.1.1 EXP
2.1.1.2 SHAPE
2.1.1.3 NAT
2.1.1.4 SIGNED_NAT

3 TDF: Optimisations

1 Introduction

1.1 TDF: Scenario of Use

TDF is an intermediate format for distributing software applications. It can be produced from a very wide range of programming languages. For expository purposes, the TDF definition is divided into three levels, referred to as Levels 0, 1 and 2.

- Level 0 is suitable for production from ANSI C (and hence any language that can sensibly be translated into ANSI C).
- Level 1 contains more features and is suitable for a wide range of languages that do not mandate garbage collection.
- Level 2 contains the full expressive power of TDF, including provision for garbage collection.

Section §2 is structured as a definition of Level 0, followed by sections that describe the additional constructs needed for Levels 1 and 2. There is no implication that separate translators must be provided for the different Levels. A translator for Level 2 should generate as efficient machine code for TDF programs that contain only Level 0 constructs, as a translator that is purpose-built for TDF Level 0. The separation is for expository purposes only, although it does indicate a possible upgrade path for TDF software (starting at support for Level 0 and being incrementally enhanced to support first Level 1 and finally Level 2).

TDF is defined in the form of a data-structure which can be thought of as an abstract syntax for programs. It contains sufficient information to allow efficient machine code to be generated from it for any computer architecture on which the software is intended to be run. For transmission, TDF is converted into a linear stream of bits. The encoding of this stream of bits is both space efficient and extensible so as to allow upwards compatibility for any future enhancements or amendments to the TDF definition.

TDF can be used for distributing "shrink-wrapped" software. To do this, software vendors produce a single version of their product in TDF. The software that produces the TDF for distribution is called a TDF producer. The largest single component of the producer is likely to be the program that converts a program written in a high-level language, such as ANSI C, into TDF. We refer to this as the **compiler** component of the producer. Once encoded, the TDF is then shipped to any of a number of target computers owned by a software purchaser. The software that converts the encoded TDF into an executable program on a target is referred to as an installer. The largest single part of the installer will be the program that generates machine code from arbitrary TDF programs. We refer to this as a TDF **translator**.

1.2 TDF: Level of Definition

TDF constructs are generalisations of the constructs found in different programming languages. This allows TDF to be the target of compilers for most programming languages. The set of TDF constructs is designed to satisfy the following requirements:

- All the information that a programming language can represent which helps a code generator produce efficient code should be representable in TDF. This means that programs distributed in TDF can be as efficient as if they were compiled with the best compiler on any target.

- Commonly provided hardware features should be easy to use - for instance, the single instruction "array and bound check" provided by many machines.

- As many optimisations as possible should be expressible as TDF to TDF transformations. This means that these optimisations can be written portably. They might be universal (i.e. beneficial for all languages and all target machines), in which case they could be included in a general-purpose TDF to TDF optimiser; they might be language specific, in which case they could be included in any of the compiler components for that language; or they might be specific to a class of architectures, in which case they could be included in translators for that class of target.

To satisfy these requirements TDF has been designed as a wide-spectrum interface, which at its highest level generalises high-level programming languages, whilst at its lowest level generalising assembler codes.

1.3 Values within a TDF System

Programming languages have always had the notion of static and dynamic values. Static values were those known at compile-time whilst dynamic values were calculated at run-time. The situation in TDF is similar. We will use the term "static" to describe values known at translate-time and "dynamic" to describe values which are calculated at run-time. (Note that in ANSI C the term "static" has a different meaning.)

1.3.1 Dynamic Values

We will start by considering run-time values. In programming languages, run-time values tend to be classified by a type system. Types are used for three different purposes in programming languages. Firstly, types help the programmer to model data in as natural a way as possible by providing a system of convenient data-structures - records, arrays etc. Secondly, types allow many structural

programming errors to be detected at compile-time. Lastly, types provide information to a compiler which helps it to generate efficient machine-code.

The TDF analogues of types are SHAPEs. They serve only the last of the three purposes described above - providing the information which translators need in order to achieve efficient memory management for any programming language on target architectures. SHAPEs are therefore designed to provide an architecture neutral abstraction of memory management making no assumptions about the properties of targets (word length, alignment constraints etc.).

1.3.2 Static Values

Apart from run-time values, there is another set of values in this TDF definition. These are the pieces of TDF program themselves, which are output by compilers. These TDF values are classified into their own system of categories which we refer to as SORTs. SORTs are analogous to the syntactic classes found in high level programming languages - identifiers, expressions, types etc. For instance, SHAPE is one of the SORTs. (To say that SHAPE is one of the SORTs means that there are pieces of TDF program which provide symbolic information about the different classes of run-time value.)

As well as SHAPE, there are twelve other SORTs.

All pieces of TDF program, whatever SORT they are, are by definition static (ie. known at translate-time). Values generated by program, whatever SHAPE they are, are in general dynamic (ie. known only at run-time). However, it may sometimes be possible to evaluate run-time expressions at translate-time, in which case they are static after all and may offer opportunities for optimisation.

1.3.3 SORTs and SHAPEs: an Example

The treatment of integers provides a good example of the relation between SORTs and SHAPEs. Pieces of TDF program which when evaluated at run-time will generate values are of the SORT EXP. (EXP stands for 'expression'.) Each EXP can be characterised by the SHAPE of the value which it will generate. For instance, an EXP which will generate an integer value is said to have an INTEGER SHAPE. Values of this SHAPE can describe any run-time integer - eg. a dynamically calculated index of an array.

Pieces of program which by contrast stand for integers known at translate-time have the SORT NAT. (NAT stands for 'natural number'.) They are not EXPs which have to be evaluated in order to generate their integer values. Instead, they already are integer values. A piece of TDF program of SORT NAT can describe any compile-time known integer - eg. a statically calculated bound for trimming an array.

1.3.4 SHAPE- and SORT-correctness

TDF relies on the programming language compiler to determine to what extent the SHAPE-correctness of programs is enforced. (An example of SHAPE-incorrectness would be the multiplication of two POINTERs.) Strongly typed languages will naturally produce SHAPE-correct programs.

Likewise, the SORT-correctness of the TDF produced by a compiler is dependent on the correctness of the compiler implementation. Neither SORT-correctness nor SHAPE-correctness need be checked by a TDF translator.

1.4 Identification of Values

TDF provides two different methods of identifying values by "names", one static and one dynamic. Identifiers which statically identify pieces of TDF program are called TOKENs. They loosely correspond to ANSI C's parameterised macros but are a great deal more powerful. Identifiers in TDF program that dynamically identify run-time values have the SORT TAG. These identifiers correspond to the names of variables and procedures in programming languages such as ANSI C.

TDF identifiers, be they TAGs or TOKENs, do nothing more than set up name/value correspondence. All the syntactic sugar associated with identifiers in programming languages - the use of mnemonic identifiers, the complexities of overloading and hiding - is provided solely to aid the human readability of programs. It provides no information which assists in the production of efficient machine code and hence has no relevance to TDF. All such syntactic sugar is eliminated by compilers to TDF.

If an occurrence of an identifier (TAG or TOKEN) is local to a program being distributed in TDF, a static integer (of SORT NAT) is used to represent it. If however the identifier identifies a value that is not purely local to the program - eg. it is a TAG standing for a system procedure, or a TOKEN whose definition is part of a library of commonly used TDF program fragments - then a value of SORT UNIQUE is used. A value of SORT UNIQUE is a truly unique identifier so that no unintentional identifier clashes can occur. As with Ethernet addresses, the uniqueness is ensured by a distributing authority which gives an organisation a seed for a sequence of unique identifiers. The representation of a value of SORT UNIQUE is a pair of integers of SORT NAT, the first being the seed and the second being the sequence number.

TAGs are an obvious generalisation of identifiers in programming languages. But TOKENs are a concept devised specifically to handle the issues that arise when software is distributed via an ANDF, as opposed to being compiled and translated on a single machine.

1.5 TDF: Architecture Neutrality

The achievement of complete architecture neutrality has been the first priority in designing TDF. The slightest shortfall from this goal would completely undermine

its usefulness as a software distribution format. This section explains how TDF allows target-dependent features of programming languages - notably ANSI C - to be completely factored out of producers and dealt with exclusively in each architecture's installer.

This complete separation of concerns means that a producer can be used to produce TDF for installation on any architecture with no alteration whatsoever.

1.5.1 SHAPEs SIZEs and OFFSETs

The design of SHAPE constructs which provide a totally symbolic description of the representation of run-time values is a central issue in the design of TDF and so warrants a detailed explanation in this section.

1.5.1.1 TUPLE

To begin, consider the C structure:

struct{unsigned char c; double f;}

When compiled to TDF, a value of this C-type will probably have TDF SHAPE:

TUPLE(unsigned_char, double)

(TUPLE is TDF's SHAPE construct describing cartesian products.) *unsigned_char* and *double* are TOKENs with definitions such as:

unsigned_char = INTEGER(0, 255)
double = FLOAT(2, 56, 0, 8)

A TDF translator will allocate space consistent with the TOKEN definitions - probably one byte for *unsigned_char* and eight bytes for *double*.

The TUPLE SHAPE shown above is a straightforward rewriting of the C type. When compiling to TDF one cannot afford to throw away the information that the value is a structure. The reason is that different architectures have different placement and alignment rules which must be obeyed. Without the knowledge that one was dealing with a structure, correct and efficient translation to machine code in this case would be impossible. For example, on a machine which had no restriction about accessing words or floating point numbers at odd byte boundaries one could compactly represent this structure in 9 bytes; a less liberal one which favoured word addressing might need 3 bytes of padding after the c-field, so requiring 12 bytes in total; and a really illiberal one might require 16 bytes by insisting that doubles start on 8-byte boundaries.

The other SHAPE constructs, besides TUPLE, which require compound information to describe the symbolic representation are UNION (§1.5.1.2), NOF (§1.5.1.3.1), SOME (§1.5.1.3.2), SIZE (§1.5.1.4.1) and OFFSET (§1.5.1.4.2). They are described in the following sections.

1.5.1.2 UNION

The UNION construct describes a representation which can hold one of a set of given SHAPEs. The TDF UNION is usually understood by a translator as specifying that the contiguous area of space required for the UNION is the maximum of the component areas, taking into account, once again, any alignment constraints.

1.5.1.3 Arrays: NOF and SOME

TDF's SHAPE system distinguishes between arrays whose number of elements are known at translate-time and those whose number of elements are not known until run-time. The statically sized arrays are described by the SHAPE construct NOF, and the dynamically sized by SOME.

1.5.1.3.1 NOF

The NOF (pronounced 'en-of') construct describes the replication of a SHAPE a translate-time known number of times. As with TUPLE and UNION, alignment constraints arise in connection with the NOF construct, with padding possibly being required between consecutive elements.

An example of the use of NOF is given by the C declaration:

unsigned char s[65536]

which would map to the SHAPE:

NOF(unsigned_char, 65536)

1.5.1.3.2 SOME

A SHAPE constructed using SOME describes a run-time known replication of a given SHAPE. The same issues of alignment and padding arise as with NOF.

For example:

SOME(double)

describes an array of *doubles*. The number of *doubles* which the array contains, and therefore the space requirements for such a value are not known at translate-time.

In order to allow efficient memory management, there are many places where TDF requires a SHAPE to be statically determinable - ie. its space requirements must be calculable at translate-time. For instance, a SHAPE used in a declaration must be statically determinable. Because of the run-time replication occurring in a SOME, a statically determinable SHAPE can contain no SOMEs unless they are hidden behind a POINTER. (The assumption is that the representation of a POINTER to a SOME is the same regardless of the number of elements in the SOME.)

The use of SHAPEs constructed by SOME is therefore restricted to constructs which produce POINTERs and which deliver run-time SIZEs and OFFSETs as described in the following section.

1.5.1.4 SIZE and OFFSET

We have established the need for retaining detailed information about values' SHAPE in TDF using TUPLE, UNION, NOF and SOME. Two further SHAPE constructs - SIZE and OFFSET - complete the picture, providing the calculation of sizes and offsets required by ANSI C and other languages.

The sizes of values and their offsets within structures or arrays are explicitly manipulated in ANSI C and also aid the implementation of other high level languages such as Ada. Sizes and offsets are quintessentially target-dependent and so TDF treats them in a totally symbolic manner.

1.5.1.4.1 SIZE

The TDF construct *shape_size*, when applied to a SHAPE, X, delivers at run-time a value which is the size of X on the host architecture:

shape_size(X)

The SHAPE of the resulting value is SIZE(X). It is important to note that its SHAPE is **not** INTEGER, measuring X's size in (say) bytes. In fact, TDF does not allow SIZEs to mix with values of other SHAPEs in ways that could compromise its architecture neutrality. For instance, no constructs are provided to add mixtures of SIZEs and INTEGERs. The reasoning behind this design decision becomes clear when we consider the generation of space and the creation of a POINTER to it.

The TDF construct *generate* generates space and creates a POINTER to it. It takes as its argument a value of SHAPE SIZE(X):

generate(shape_size(X))

Space for a value of SHAPE X is generated and a POINTER to it delivered. The important point here is that the number of bytes required is not determined explicitly

by the compiler to TDF and stated as an INTEGER. To allow this would be to compromise the architecture neutrality of TDF. Instead it is provided at run-time by *shape_size*, ensuring that the host architecture's placement and alignment rules are respected.

(Although the result of *shape_size(X)* is in general only available at run-time, it is determinable at translate-time if *X* contains no SOME constructs unhidden by POINTERs.)

1.5.1.4.2 OFFSET

Just as SIZEs are motivated by the need to supply a completely architecture neutral argument to *generate* and related operators, the SHAPE construct OFFSET is needed in order to achieve completely architecture neutral pointer arithmetic.

The TDF construct *array_element_offset* (§2.1.3.7.6), when applied to a SHAPE, X, delivers a value which is the distance between elements in a array (formed by SOME or NOF) of elements of SHAPE X on the host architecture:

array_element_offset(X)

The SHAPE of the resulting value is OFFSET(X,X), to be understood as the offset between two adjacent values of SHAPE X. As with *shape_size* in the previous section, it is important to note that its SHAPE is **not** INTEGER. TDF does not allow OFFSETs to mix with values of other SHAPEs. For instance, no constructs are provided to add mixtures of OFFSETs and INTEGERs.

OFFSETs find application in pointer arithmetic operations. For instance, the construct *add_to_ptr* takes as its arguments a POINTER and an OFFSET:

```
add_to_ptr(p,
           array_element_offset(X)
          )
```

A new POINTER is delivered which points to a space one element removed from that pointed to by *p*, assuming *p* to have been pointing at an array of values of SHAPE X. The important point here is that the number of bytes' displacement is not determined explicitly by the compiler to TDF and stated as an INTEGER. To allow this would be to compromise the architecture neutrality of TDF. Instead it is provided at run-time by *array_element_offset*.

(Although the result of *array_element_offset(X)* is in general only available at run-time, it is determinable at translate-time if *X* contains no SOME constructs unhidden by POINTERs.)

OFFSETs are also of relevance to TUPLEs. An OFFSET(X,Y) is the OFFSET from

the start of X to the start of Y in a TUPLE (X,Y). The offset of the *j*-th element from the start of a TUPLE(S_1,S_2 .. S_j ..) is given by:

tuple_element_offset(TUPLE(S_1,S_2 .. S_{j-1}), S_j)

and has SHAPE OFFSET(TUPLE(S_1,S_2 .. S_{j-1}),S_j). (See §2.1.3.7.7.)

Unlike SIZEs, OFFSETs are additive under TUPLEing. Consider, for instance, a TUPLE(a,b,c). If *off1* is the OFFSET from the start of the *a* field to the start of the *b* field, and *off2* is the OFFSET from the start of the *b* field to the start of the *c* field, then *off1* + *off2* is the OFFSET from the start of the *a* field to the start of the *c* field.

Contrast this with the behaviour of SIZEs:

SIZE(a) + SIZE(b) + SIZE(c)	does not necessarily equal	SIZE(TUPLE(a,b,c))

1.5.1.5 SHAPEs SIZEs and OFFSETs: ANSI C

The ANSI C notion of "size" corresponds to a TDF OFFSET between elements of an array. ANSI C has no explicit notion corresponding to the TDF concept of SIZE, but an ANSI C to TDF compiler will use the SHAPE constructs SIZE and OFFSET in implementing C.

ANSI C deals only in fixed size objects or repetitions of fixed size objects. Hence if SIZE(Z) occurs in the TDF output of an ANSI C compiler then either Z contains no SOME constructs or else it is of the form SOME(X) where X contains no SOME constructs. This means that in TDF derived from ANSI C all SIZEs and OFFSETs are constant values and can be determined at translate-time, allowing optimisation.

An account of SHAPEs, SIZEs and OFFSETs as they relate to Ada is given in §2.2.2.3

1.5.2 Conditional Compilation

TDF has been designed to meet programming languages' requirements with regard to conditional compilation. In brief, what is required is the ability to choose, at translate-time, which of two different program fragments to translate.

For the purposes of ANSI C, only two of the TDF SORTs need be conditionally translated in this fashion - EXP and VARIETY. These cover evaluable pieces of program and the determinants of the sizes of integers. Two constructs - *exp_cond* and *variety_cond* - express their conditional compilation. Each supplies one or other of a

pair of program fragments depending on the value delivered by an expression intended to be evaluated at translate-time. (See §2.1.5 for further detail.)

Similar constructs providing conditional compilation for all the other SORTs form part of Level 1 and are described in §2.2.4.

1.5.3 Tokenisation

A TOKEN identifies a (possibly) parameterised construct whose result can be of any SORT. The definition of the TOKEN (in procedure or macro terms, its body) can be supplied at a number of different times in the production or installation process:

- The TOKEN definition can be supplied by the producer, in which case that same definition will be distributed to all targets. A typical such usage would be to make a commonly occurring piece of TDF into a token definition in order to compress the amount of TDF distributed. The substitution of the definition for the token will occur in the installer and will be performed by macro expansion.

  The producer can supply a TOKEN definition in two ways. A TOKEN can be defined over the whole of a unit of TDF being distributed - called a TDF capsule (see §1.6 for details). This is done by including its definition in the list of TOKEN definitions which form part of the capsule. Alternatively, a TOKEN can be defined over a delimited piece of TDF program using *tokenise* (see §2.1.4).

- A TOKEN's definition might be supplied by the installer. There are a number of usages of this:

  - a piece of TDF might be used so frequently that its definition is supplied by all installers. This is a similar usage to the one above but eliminates the need to distribute the even the TOKEN's definition, which compresses the TDF even more.

  - the TDF definition substituted for the TOKEN may be target specific e.g. the datastructure used by a print procedure. (The TOKEN definition for this particular datastructure would need no parameters.)

- The TOKEN may be recognised by the translator and implemented directly. There are a number of uses for this approach:

  - A TOKEN might be used to represent a construct such as vector inner product. A producer might supply a portable definition of this TOKEN. However an installer on a machine such as a CRAY might choose to ignore the portable definition

and make full use of the CRAY's parallelism in implementing the vector inner product.

• if a new language were invented requiring a new feature to be added to TDF, it could be defined as a TOKEN which installers implemented according to its definition.

• The TOKEN might be bound during linking to an external function that has been precompiled from a programming language, or directly written in assembler. The mechanism for doing this is defined as part of the installation process.

1.6 Structure of a TDF Capsule

The unit of encoded TDF that is distributed is called a TDF "capsule". A TDF capsule consists of:

• information to control linking with other TDF capsules and with the host system.

• information setting limits on the number of TAGs used in the capsule and other data which help installers run as efficiently as possible.

• a set of TOKEN definitions. The order of the definitions is not significant, since all the TOKENs are visible in all the definitions. Not all the TOKENs used in the capsule need be defined here. Those that are not will be supplied by the time translation occurs.

• a set of TAG introductions. A TAG introduction contains all the information about the run-time value identified by a TAG except its actual definition. A TAG introduction is required for all TAGs that are global to the capsule. (TAGs that are local to procedure bodies do not need introductions.)

All TAG introductions contain the SHAPE of the value being identified and an indication of whether the TAG is local to the piece of TDF being distributed or is external - for example, possibly being shared with other TDF capsules being distributed. If the TAG is external, its definition may be supplied in this piece of TDF or it may have to be found elsewhere by the installer. If a TAG is external then a string of characters is provided which can be used by the installer, typically to link to a pre-compiled routine on the target. TAG introductions also contain information specific to the particular kind of TAG declaration (eg. identity declaration, variable declaration - see §2.1.3.1).

As with TOKEN definitions, the order of TAG introductions is not significant.

- a set of TAG definitions. A TAG definition associates a TAG introduction with the TDF definition of a piece of program (of SORT EXP) which when evaluated produces a run-time value.

  The order of TAG definitions is not significant.

  Optionally one TAG may be identified as the main TAG. This TAG identifies a piece of program whose evaluation corresponds to the main body of the program being distributed.

(The contents of a TDF capsule are described in more detail in §4.2.)

1.7 TDF Terminology

1.7.1 Specifying Translator Behaviour

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

- "undefined": means that the translator 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 translator must perform P.

- "should": when the phrase "P should be done" (or similar phrase involving "should") is used, translators are advised to perform P, and compiler writers may assume it will be done if possible. This usage usually relates to optimisations which are being advised.

- "will": when the phrase "P will be true" (or similar phrases involving "will") is used, the translator may assume that P holds without attempting to check it. If, in fact, a compiler has produced TDF for which P does not hold, the effect is undefined.

- "target-defined": means that there is a definition but this varies from one target machine to another. Each target translator shall define everything which is said to be "target-defined".

1.7.2 Describing Program Construction

For transmission, TDF is compactly encoded as described in §4.3. Though optimal for the purposes of transmission by machine, the encoded form of TDF is not a

convenient medium for describing the structure of TDF program to the human reader. We therefore use the following notation for the remainder of this document:

Some SORTs consist of a fixed number of named alternatives. To indicate a particular alternative, we simply write its name. For instance, the two alternatives for BOOL appear as:

TRUE

FALSE

Other SORTs consist simply of integers or a subset of integers which can be written down in the usual way, eg.:

3

Certain SORTs can consist of a tuple of components (ie. they are Cartesian products of other SORTs). To write these down we list their components. For instance, a VARIETY may consist of a pair of NATs:

(0,255)

The SORTs EXP and SHAPE are recursively defined, with a considerably richer set of primitives and constructs than the other SORTs have. All these primitives and constructs are set out in §2.1, §2.2 and §2.3.

Primitive EXPs are simply named, as in:

top

The application of an EXP construct is denoted as follows:

goto(2,
    top
)

In text, the names of EXP constructs will appear in lower case italics.

The construct *goto* takes two arguments. In this case, the first, a LABEL, is simply a NAT. The second is the primitive EXP, *top*.

As with EXPs, primitive SHAPEs are simply named, as in:

BIT

And the application of a SHAPE construct is denoted as follows:

NOF(BIT,100)

In text, the names of SHAPE constructs will appear in upper case. (The reader may already have noticed that the names of the SORTs also appear in upper case - as does the word SORT.)

The construct NOF takes two arguments, a SHAPE and a NAT. In this case, the first is the primitive SHAPE BIT, and the second is a NAT.

Since the SHAPEs of values produced when EXPs are evaluated are important, we generally state the SHAPE when specifying TDF constructs. For instance, an EXP which evaluates to produce a value of SHAPE BIT is described as an EXP BIT. TAGs are likewise qualified.

The following example, drawn from §2.1.3.2.21 which specifies the construct *truncate*, shows how the EXP and SHAPE notations look in practice:

truncate
ov_err:ERROR_TREATMENT,
v:VARIETY,
arg:EXP FLOAT(F)

-> EXP INTEGER(v)

The construct's arguments (if any) precede the -> and the result follows it. Each argument has the form:

name:SORT

The name standing before the colon is for use in any English description which may accompany the notation.

The example given above indicates that *truncate* takes three arguments. The first argument, *ov_err*, has SORT ERROR_TREATMENT. The second, *v*, has SORT VARIETY and defines the VARIETY to be used to construct the integer SHAPE in the result of *truncate* (see below). The third argument, *arg*, is an expression of SORT EXP, and as mentioned before we append the SHAPE of the EXP, FLOAT(F). *arg* is the piece of program which will deliver the floating point number to be truncated.

After the -> comes the SORT of the result of *truncate*. The result is an EXP INTEGER(v) - a piece of program which, when evaluated, will deliver a value whose SHAPE is INTEGER(v), the truncated floating point number.

No account is given here of the dynamic semantics of *truncate*, only its static semantics - ie. the SORTs and SHAPEs of its arguments and result and the relations between them. However, when each of the EXP and SHAPE primitives and constructs is set

out later in this document, an English account of its dynamic semantics accompanies it.

The format for the description of the construction of a SHAPE is similar. For instance, the SHAPE construct SIZE:

SIZE
sh:SHAPE

-> SHAPE

takes one SHAPE argument, which for the purposes of any accompanying English text is named *sh*, and yields a SHAPE result.

Three further conventions are needed in order to express TDF constructs. Some constructs have a variable number of parameters. For instance, *make_tuple* (§2.1.3.9.1) can be used to make up tuples with any number of components (>1). We write this as:

$$\Pi_{i=1}^{n} S_i$$

The symbol "Π" is chosen to indicate cartesian product; i ranges from 1 to n; and the S_i are the components. In addition it is necessary sometimes to add qualifying predicates, which we enclose in curly brackets, as in

$$\Pi_{i=1}^{n} S_i \quad \{ n>1 \}$$

Some constructs have parameters which may be optionally be omitted. To indicate this we enclose the SORT of the optional parameter in brackets and apply a postfix _OPTION, e.g.

(TAG UNTRACED_PROC)_OPTION

meaning either a TAG UNTRACED_PROC or nothing.

2 Definition

2.1 TDF Level 0

This section defines the SORTs, SHAPEs and EXPs which go together to form TDF Level 0 - ie. the subset of full TDF which is required in order to implement ANSI C.

2.1.1 Level 0 SORTs

There are thirteen SORTs in TDF:

EXP	UNIQUE
SHAPE	TAG
NAT	LABEL
SIGNED_NAT	NTEST
VARIETY	ERROR_TREATMENT
FLOATING_VARIETY	EXCEPTION_HANDLER
BOOL	

All form part of TDF Level 0, apart from EXCEPTION_HANDLER, which appears in Level 1. Each of the Level 0 SORTs is described below:

2.1.1.1 EXP

EXP is short for 'expression'. This is the main SORT in TDF. It describes a piece of program that generates and manipulates run-time values. A substantial part of this document (§2.1.3, §2.2.3 and §2.3.3) is taken up with descriptions of the TDF constructs that are used to create EXPs. The definitions of many EXPs are recursive; EXPs are built up from sub-EXPs and values of other SORTs. There are constructs delivering EXPs that correspond to the declarations, program structure, procedure calls, assignments, pointer manipulations, arithmetic operations, tests etc. of programming languages.

The types (in programming language terms) of the run-time values generated and manipulated by EXPs are described by the SORT SHAPE:

2.1.1.2 SHAPE

SHAPEs give TDF translators symbolic size and representation information about run-time values. Values of the same SHAPE will be represented in the same way and occupy the same amount of memory at run-time on a given architecture.

The definition of SHAPEs is recursive and is built up from a set of basic SHAPEs

such as BIT and PROC and constructs for compound datastructures such as tuples, arrays (both statically and dynamically sized), pointers and unions. The constructs for forming SHAPEs are described in §2.1.2, §2.2.2 and §2.3.2.

2.1.1.3 NAT

A value of SORT NAT is a static non-negative integer value of unbounded size.

2.1.1.4 SIGNED_NAT

A value of SORT SIGNED_NAT is a static integer value, positive or negative, of unbounded size.

2.1.1.5 VARIETY

A value of SORT VARIETY describes the different kinds of integer which are available at run-time. It is either a tuple of two natural numbers of SORT SIGNED_NAT which describe the lower and upper bound of integers that must be representable by the integer value at run-time (as discussed in §2.1.2.3.1), or it can be one of four specially distinguished VARIETYs.

(SIGNED_NAT, SIGNED_NAT) |
best_signed |
best_unsigned |
unlimited_signed |
unlimited_unsigned

(*unlimited_signed* and *unlimited_unsigned* do not appear in Level 0, but form part of Level 2. They are included here for completeness.)

2.1.1.6 FLOATING_VARIETY

A value of SORT FLOATING_VARIETY describes the kinds of floating point numbers which are available at run-time. It is a tuple of four values of SORT NAT.

(NAT, NAT, NAT, NAT)

These give details about the base to be used, the number of digits that must be representable in the mantissa and the minimum and maximum numbers that must be representable by the exponent (see §2.1.2.3.2 for further details).

2.1.1.7 BOOL

A static value of SORT BOOL is either true or false.

true |
false

2.1.1.8 UNIQUE

As discussed in §1.4 UNIQUE is composed of two values of SORT NAT.

(NAT, NAT)

The first NAT identifies the issuer, the second is a sequence number, ensuring uniqueness among the UNIQUEs issued by a particular issuer.

2.1.1.9 TAG

A value of SORT TAG is an identifier standing for a run-time evaluated expression of SORT EXP. It is represented by a value of SORT NAT or of SORT UNIQUE (as described in §2.1.3.1).

NAT |
UNIQUE

Each TAG has an associated SHAPE which is defined in the TAG's introduction (as described in §4.2).

2.1.1.10 LABEL

A LABEL identifies a piece of program and serves the role of labels in traditional languages and hardware architectures, ie. a destination for jumps. A LABEL may either be a NAT or a UNIQUE.

NAT |
UNIQUE

A NAT is used for LABELs which are private to a TDF capsule, but for LABELs which are required to be accessible between capsules, UNIQUEs are used.

2.1.1.11 NTEST

A value of SORT NTEST identifies one of a number of arithmetic tests. There are six NTESTs available.

greater_than |
greater_than_or_equal |
less_than |
less_than_or_equal |
equal |
not_equal

The names are self-explanatory.

2.1.1.12 ERROR_TREATMENT

A value of SORT ERROR_TREATMENT controls program behaviour in the event that a run-time error occurs. That behaviour can be one of four named possibilities, or it can be the passing of control to a LABEL.

impossible |
ignore |
standard_exception |
standard_signal |
LABEL

The significance of the different possible ERROR_TREATMENTs is as follows. (*standard_exception* forms part of Level 1 and is described in §2.2.1.2.)

2.1.1.12.1 Impossible

This argument is used when the error cannot occur. For example, if the divide operation is dividing by a constant, which is known not to be zero, then the div0_err ERROR_TREATMENT should be given the value *impossible*. This permits the translator to avoid creating any code that might have been needed. This argument should be produced by compiler writers whenever possible, since it permits the least and fastest code to be produced.

For example, when translating an arithmetic operation with error treatment *impossible* on Vax, if the program at this point has overflow trap flag set or unset, the trap flag need not be changed.

If the error in question does nevertheless occur, the effect of the operation is undefined.

2.1.1.12.2 Ignore

This argument is used when the error can occur, but an attempt is to be made to

carry on. In some operations the effect will be undefined, in others a definition is given.

For example, when translating an arithmetic operation with error treatment *ignore* on Vax, if the program at this point has overflow trap set, it will have to be unset.

2.1.1.12.3 Standard Signal

This argument is used when an error can occur and the desired effect is to call a signal procedure (see §2.1.3.11) as if from the construct being evaluated. For each standard exception there is a signal procedure. All the signal procedures which can be called from an ERROR_TREATMENT shall produce *bottom* (see §2.1.2.1.1 for an account of *bottom*); that is, they can only be left by a long jump or by raising an exception.

ANSI C uses this method of handling errors.

Cross-reference: Exceptions: Discussion §2.2.3.2.3, Signals: Discussion §2.1.3.11, bottom §2.1.2.1.1

2.1.1.12.4 LABEL ERROR_TREATMENT

This argument is used when an error can occur and the desired effect is to transfer control to a LABEL. The LABEL is part of this argument. The LABEL will expect the POINTER value described in §2.2.3.2.3.1. This ERROR_TREATMENT can only be used where the LABEL is available.

Compilers which are processing exception-like constructs, for example in Ada, and know at compile-time which exception handler will be used, should introduce a LABEL and use the *error_label* argument instead of the *standard_exception* argument.

Cross-reference: LABEL §2.1.1.10, top §2.1.2.1.2, Exceptions: Discussion §2.2.3.2.3, Availability of LABELs: Discussion §2.1.3.5.1, Jumping with Values: Discussion §2.1.3.5.2

2.1.2 Level 0 SHAPEs

2.1.2.1 Level 0 Basic SHAPEs

2.1.2.1.1 BOTTOM

BOTTOM is the SHAPE associated with constructs that do not terminate normally, such as signal procedures. Such a construct can only be left by a long jump or by raising an exception.

2.1.2.1.2 TOP

TOP is the SHAPE associated with constructs that return no useful value. For example when a TDF construct jumps to a LABEL but does not jump with any value, it jumps with a value of SHAPE TOP.

2.1.2.1.3 BIT

BIT is the SHAPE describing values which have only two possible conditions.

2.1.2.1.4 UNTRACED_PROC

UNTRACED_PROC is the SHAPE describing any procedure value in a non-garbage collecting system, and those procedure values in a garbage collecting system which reside in the untraced kernel. (§2.3.2.1.1 provides an account of the Level 0 and 1 untraced kernel.) The only ultimate use that can be made of an UNTRACED_PROC is to apply it to a parameter.

Equality of representation is undefined for UNTRACED_PROCs.

UNTRACED_PROCs are values which may have a limited lifetime. (The concept of lifetime is introduced in §2.1.3.12.)

Cross-reference: procedures §2.1.3.6

2.1.2.1.5 LABEL_VALUE

LABEL_VALUE is the SHAPE describing values which represent LABELs. A LABEL_VALUE can be created by *make_label_value*. It is used to implement language features which need to manipulate LABELs as values, such as the long jump of C and PERFORM in COBOL. A LABEL_VALUE is a value with a limited lifetime.

Equality of representation for LABEL_VALUEs is not defined.

Cross-reference: make_label_value §2.1.3.5.11, Lifetimes: Discussion §2.1.3.12

2.1.2.2 Least Upper Bound

Every TDF construct producing an EXP specifies the SHAPE of that EXP. The EXP's SHAPE may always be the same or it may depend in some way on the arguments supplied to the construct. For instance, *make_int* (§2.1.3.2.2) produces an EXP of SHAPE INTEGER(V), where the VARIETY of the integer is governed by the VARIETY which is supplied as one of the construct's arguments.

Certain constructs produce EXPs which at run-time deliver a value derived from the

evaluation of one of a number of argument EXPs, which one being undetermined at translate-time. In these cases, the SHAPE of the result is deemed to be the Least Upper Bound or LUB of the SHAPEs of all the EXPs which could provide the result. See the specification of *case* (§2.1.3.5.4) for an example of this.

The rules governing the calculation of the LUB of SHAPEs are as follows:

bottom LUB x = x

top LUB x = top

x LUB x = x

if neither x nor y are bottom or top and x≠y, then x LUB y = top

(The names *bottom* and *top* are chosen because the SHAPEs form a semi-lattice.)

The LUB of a number of SHAPEs is denoted:

$$\mathrm{LUB}_{i=1}{}^{n} X_i$$

where it is understood that:

$$\mathrm{LUB}_{i=1}{}^{n} X_i = X_1 \,\mathrm{LUB}\, X_2 \ldots \mathrm{LUB}\, X_n$$

LUB features in the descriptions of some of the compound SHAPEs which follow.

2.1.2.3 Level 0 Compound SHAPEs

Compound SHAPEs are SHAPEs which are not primitive, in the sense that the constructs which form them take arguments.

Circular SHAPEs can be constructed using *tokenise* (see §2.1.4). Eg.

```
tokenise(list,
        SHAPE,
        TUPLE(BIT, PART_POINTER(list))
       )
```

However no SHAPEs will be constructed whose memory requirement is infinite.

2.1.2.3.1 Integer SHAPEs

Most of the integer arithmetic operations - *plus*, *minus* etc. - are defined to work in the same way on different kinds of integer. If these operations have more than one

argument, the arguments have to be of the same kind, and the result is also of this kind.

The different kinds of integer are called different VARIETYs. (The SORT VARIETY was introduced in §2.1.1.5.) These VARIETYs fall into two classes. The first class comprises VARIETYs which can only express a bounded number of different integers. These are called limited VARIETYs. Some of the operations on integers with limited VARIETYs can cause overflow errors. Only the limited VARIETYs are used by C.

(The other class of VARIETYs, unlimited VARIETYs, appears in TDF Level 2 and is described in §2.3.2.4. They are used to represent integers of unbounded size.)

Operations which can cause overflow take an ERROR_TREATMENT argument which specifies how an overflow error is to be treated. These operations are applicable to limited and unlimited integers alike.

The represention of limited integers are equal if and only if the integers are equal. Equality is only defined for identical VARIETYs.

SHAPEs describing integers are constructed by the SHAPE construct INTEGER, taking a value of SORT VARIETY as its argument. Thus:

INTEGER(0,255)

is a SHAPE describing an integer value whose VARIETY is (0,255), specifying that it may lie between 0 and 255 inclusive, and for which a translator can accordingly plan space.

Limited VARIETYs in which the least limit is less than zero are known as signed VARIETYs. Limited VARIETYs with the least limit greater than or equal to zero are known as unsigned VARIETYs.

In addition to the limited VARIETYs whose bounds are specified, two limited VARIETYs are provided which serve the same purpose as ANSI C's *int* and *unsigned int*. They are named *best_signed* and *best_unsigned*. Integers having these VARIETYs shall have values at least including the range 1-2^15 to (2^15)-1 and 0 to (2^16)-1 respectively, but otherwise defined to be the most appropriate for the target machine.

When any operation delivering an integer belonging to a limited VARIETY produces a result not lying between the bounds of that VARIETY, an integer overflow error occurs. Every operation which can produce such a result has an ERROR_TREATMENT argument which specifies how this error is to be dealt with.

The LUB of two limited VARIETYs is *top* unless both the least and greatest requested

limiting values are the same, or both are the same named VARIETY. Thus no assumption is made about the size of words in the target.

Cross-reference: maxint §2.1.3.2.16, minint §2.1.3.2.17, Least Upper Bound §2.1.2.2

2.1.2.3.1.1 Recommendations about Integer VARIETYs

Three recommendations are made about the use of integer VARIETYs.

- First recommendation: when SIGNED_NATs are chosen to define a limited VARIETY, their values should reflect as precisely as possible what is needed by the program. This choice should not be influenced by knowledge of what is available on common machines (except where the purpose is specifically to take advantage of such knowledge). It is the task of the TDF translator to make intelligent decisions. Again the use of integers for indexing provides an example where the translator should be allowed as free a choice of representation as possible.

- Second recommendation: whenever it is reasonable, the VARIETYs *best_signed* or *best_unsigned* should be used. (Other VARIETYs are likely to be more expensive in terms of explicit overflow checking.) The assumption should never be made that the *best* VARIETYs will be a 32 bit integers. It is important for translators to be able to make *best* VARIETYs occupy less than 32 bits to allow for addressing and to allow for garbage collection techniques.

- Third recommendation: integer VARIETYs should be tokenised in such a way that useful selective alterations may be made purely in the target machine. (See §1.5.3 for an account of tokenisation.) It may be that certain operations involving integers can usefully be transformed to make best use of an architecture's facilities. So that the relevant integer VARIETYs can be selectively substituted, the integer arguments to these operations should belong to a particular tokenised VARIETY, and other integers to another VARIETY. (An instance of this is array indexing operations, where freedom to determine the characteristics of the integers involved on a particular target permits a choice of memory model to be made by the installer.)

2.1.2.3.2 Floating Point SHAPEs

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 also of this kind.

The different kinds of floating point number are called FLOATING_VARIETYs. (FLOATING_VARIETYs were introduced in §2.1.1.6.) SHAPEs describing floating point values are constructed by the SHAPE construct FLOATING, taking a value of SORT FLOATING_VARIETY as its argument. Thus:

FLOATING(10,30,0,100)

is the SHAPE of a floating point value of FLOATING_VARIETY (10,30,0,100). This signifies that its BASE is 10, it has 30 digits in its MANTISSA, its MINIMUM_EXPONENT is 0 and its MAXIMUM_EXPONENT is 100!

BASE is the base with respect to which the remaining numbers are given.

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

MINIMUM_EXPONENT is the required integer, n, such that BASE raised to the power $-n$ can be represented as a non-zero floating point number of the variety.

MAXIMUM_EXPONENT is the required integer such that BASE raised to that power is representable as a floating point number of the variety.

The base given need bear no relation to the base for floating point numbers in any target architecture. Commonly, as in ANSI C on all known architectures, the definition may be given in terms of a decimal base, but the implementation may be binary.

Equality of representation for floating point numbers is defined to be equality of representation in the target machine. It is therefore target-defined.

The use of a FLOATING_VARIETY in TDF expresses the intention that a correct program will only use the values implied by the requirements. 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 error may be produced.

The LUB of two FLOATING_VARIETYs is *top* unless each of their defining parameters is equal.

Any number of FLOATING_VARIETYs may be asked for by a TDF program, though it is recommended that the number should be severely limited. The space taken in the TDF for transmission of FLOATING_VARIETYs should be minimised by tokenising (§1.5.3) the required FLOATING_VARIETYs and using the TOKENs instead of the full form.

2.1.2.3.2.1 Recommendations about FLOATING_VARIETYs

Two recommendations are made about the use of FLOATING_VARIETYs in TDF.

- First recommendation: when parameters are chosen to define a FLOATING_VARIETY their values should reflect as precisely as possible what is needed by the program. This choice should not be influenced by knowledge of what is available on common machines. It is the task of the TDF translator to make intelligent decisions.

- Second recommendation: FLOATING_VARIETYs should be tokenised in such a way that useful selective alterations may be made purely in the target machine. (See §1.5.3 for an account of tokenisation.) It may be that a certain operations involving floating point values can usefully be transformed to make best use of an architecture's facilities. So that the relevant floating point VARIETYs can be selectively substituted, the floating point arguments to these operations should belong to a particular tokenised FLOATING_VARIETY, and other floating point values to another FLOATING_VARIETY.

2.1.2.3.3 POINTER SHAPEs

There are five SHAPE constructs collectively known as POINTERs. They are:

UNTRACED_POINTER
WHOLE_POINTER
SHAKY_WHOLE_POINTER
PART_POINTER
SHAKY_PART_POINTER

Only UNTRACED_POINTER forms part of Level 0. All the others are concerned with garbage collection and are introduced in Level 2 (see §2.3.2.1).

All the POINTER constructs take a SHAPE as an argument. The SHAPE describes the value to which the POINTER points. This gives TDF translators the freedom to implement POINTERs in different ways depending on the SHAPE of the values to which they point.

2.1.2.3.3.1 UNTRACED_POINTER SHAPEs

An UNTRACED_POINTER is a POINTER which points to a space allocated in the untraced kernel of a computer's memory - untraced, that is, by any garbage collector.

No TDF constructs can create an UNTRACED_POINTER explicitly. This can only be done by library routines such as *calloc* and *malloc*. Likewise, deallocation.

UNTRACED_POINTERs have equal representation if and only if they are identical - ie. they are copies of a value produced from one particular POINTER allocation.

The lifetime of an UNTRACED_POINTER depends on the manner of its creation. If it arises from a *variable* or *variable_no_init* construct, then its lifetime extends over the body of that construct. However, if it arises from explicit use of a library routine, then its lifetime is undefined.

Cross-reference: Garbage Collection: Discussion §2.3.2.1.1, Lifetimes: Discussion §2.1.3.12, sharing §2.1.3.4.1.1

2.1.2.3.4 TUPLE SHAPEs

components:$\Pi_{i=1}^{n}$ (s_i:SHAPE)

-> SHAPE

A TUPLE is a cartesian product of the SHAPEs *components*.

None of the *components* will be top.

Two TUPLEs have equal representations if their components all have equality of representation defined and if the components are pairwise equal in representation. This implies that any padding between fields (put there to satisfy alignment rules) must have standard values.

TDF requires that the fields be represented in memory in the same order as they occur in the TUPLE. That is, the OFFSET from one field to the next is always positive. More than that, TDF requires the representation of the first n fields of a TUPLE to be unaltered by adding an additional field at the end.

Note: the advantage of implementing a hierarchy of properties in this way outweighs the possible gains of a more compact representation from re-ordering the elements in the TDF translator.

2.1.2.3.5 ALIGNED_TUPLE SHAPEs

components:$\Pi_{i=1}^{n}$ (s_j:SHAPE)

-> SHAPE

Like TUPLE, an ALIGNED_TUPLE is a cartesian product of the SHAPEs *components*. All the rules which apply to TUPLEs apply to ALIGNED_TUPLEs.

The two SHAPE constructs are distinguished in order to provide for the application of a procedure to more than one parameter. The TDF construct *apply_proc* allows only

one parameter to be supplied to a procedure. In order to model programming languages which permit more than one parameter to be supplied, a TDF producer gathers the parameters into an ALIGNED_TUPLE and supplies that single value. Because some architectures have different conventions regarding the packing of procedure parameters and the disposition of ordinary structures, TDF reflects this distinction between the two ways of parcelling data so that translation to the targets can be as efficient as possible.

All the operations applicable to TUPLEs are applicable to ALIGNED_TUPLEs, except that *apply_proc* will not be used on a TUPLE parameter. ALIGNED_TUPLEs are created using *make_aligned_tuple* (§2.1.3.9.2) rather than *make_tuple* (§2.1.3.9.1).

2.1.2.3.6 UNION SHAPEs

alternatives: $\Pi_{i=1}^{n}$ (s_i:SHAPE)

-> SHAPE

A UNION value contains one of the *alternatives*, s_i. A discriminant to determine which alternative is in use is **not** a part of the value. If it is needed, such discrimination is performed elsewhere.

UNIONs have equal representation if the values present belong to the same SHAPE and are equal in representation.

2.1.2.3.7 SIZE SHAPEs

sh:SHAPE

-> SHAPE

The SHAPE of run-time values which measure amounts of memory in an architecture neutral manner. The notions of SIZE and OFFSET are closely related, but not identical. An OFFSET is a run-time value which measures the displacement between spaces holding values in an architecture neutral manner. Given a TUPLE consisting of a pair of values of SHAPEs *a* and *b*, the start of the *b* value is not necessarily displaced from the start of the pair by the SIZE of *a*. There may be memory alignment requirements which mean that there has to be some dead space between the end of *a* and the start of *b*. In general we need to know both SIZE information and OFFSET information.

In TDF, SIZE values can be converted to integer values which give the minimum number of bits of space needed to hold something (this information is needed by Ada). They can also be turned into integer values giving the minimum number of bytes of space needed to hold something (this information is needed by ANSI C).

In ANSI C the requirements on SIZEs and OFFSETs are not complex. But Ada

requires compound values which are made up from components which are of run-time determined SIZE. This, together with the need for the alignment information to be given symbolically, makes the general concept of SIZE more advanced than that required by ANSI C alone.

Translators need to know the structure of the SHAPE of which a SIZE is being computed. If they did not know, it would be necessary to carry this structure around at run-time, in order to work out the padding needed within tuples. But if the translator does have this information, it need only carry around at run-time the number of bytes (or bits) of space needed. The amount of padding can be computed at translate-time.

However, translators can only carry around general structure information; they cannot know the actual (computed) SIZEs of arrays etc. This general SHAPE information therefore uses the SOME construct, which gives enough information to do these calculations.

Note that the role of the SHAPE parameter of SIZE is not at all the same as the role of the SHAPE parameters in compound SHAPEs. A value whose SHAPE is SIZE(UNTRACED_PROC), for example, does not contain a UNTRACED_PROC value, it is merely a measure of the amount of memory occupied by a UNTRACED_PROC value. This is why the SHAPEs in this position need not be SOME-free.

SIZE(SH1) LUB SIZE(SH2)

will be *top* unless SH1 = SH2, in which case it will be SIZE(SH1). (See §2.1.2.2 for an account of LUB - Least Upper Bound.)

A more detailed account is given in the section on SHAPEs and SIZEs.

Equality of representation for SIZEs is undefined, however *size_test* may be used to compare SIZEs.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1, generate §2.3.3.1, size_test §2.1.3.7.4, SOME §1.5.1.3.2, Level 0 Compound SHAPEs §2.1.2.3, Level 1 SHAPEs §2.2.2, Level 2 SHAPEs §2.3.2

2.1.2.3.8 OFFSET SHAPEs

a:SHAPE
b:SHAPE

-> SHAPE

The SHAPE of run-time values which measure memory offsets in an architecture neutral manner. As explained in §2.1.2.3.7, given a TUPLE consisting of a pair of

values of SHAPEs *a* and *b*, the start of the *b* value is not necessarily displaced from the start of the pair by the SIZE of *a*. The need to know this displacement leads to the requirement for OFFSET values.

A value of SHAPE OFFSET(*a*,*b*) is a measure of the displacement of a value of SHAPE *b* from the start of a TUPLE(*a*,*b*). A value of SHAPE OFFSET(*a*,*a*) is a measure of the displacement between successive values in an array containing *a*'s.

Equality of representation for OFFSETs is undefined.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1, SIZE SHAPEs §2.1.2.3.7

2.1.2.3.9 NOF SHAPEs

s:SHAPE,
n:NAT

-> SHAPE

An NOF value is an array of *n* values of SHAPE *s*. *n* is known at translate-time.

NOF values have equal representations if they have the same number of components, if their components have equality of representation defined and if the components are pairwise equal in representation.

An NOF value differs from a TUPLE of *n* values of the same SHAPE in that the selection of a component by a computed index is allowed. (The construct *index* is described in §2.2.3.2.17.)

2.1.2.3.10 SOME SHAPEs

s:SHAPE

-> SHAPE

A SOME value is an array of values of SHAPE *s*. Unlike NOF, the number of elements in the array is not known at translate-time.

This SHAPE is not on the same footing as the other SHAPEs. If a value of SHAPE Z contains no value whose SHAPE is SOME, the amount of space needed to hold a value of SHAPE Z is known at translate-time. Such a SHAPE Z will be called SOME-free. Almost all the SHAPEs used in a TDF tree will be SOME-free, because translators need to know how much space to allow for a value of the SHAPE. Only values which are handled through POINTERs can have SHAPEs which are not SOME-free. Only the SHAPEs involved in calculations of SIZEs and OFFSETs need not be SOME-free. In the TDF specification every construct which

uses a SHAPE is defined either to be SOME-free or to be not necessarily SOME-free.

2.1.3 Level 0 EXPs

The EXP constructs required for implementing ANSI C can conveniently be broken down into ten broad categories:

Declarations and Naming
Integers
Floating Point Values
POINTERs
Program Structure and Flow of Control
Procedures
SIZEs
NOFs
TUPLEs, ALIGNED_TUPLEs and UNIONs
Miscellaneous

These are described in the following sections. (The notation used for describing the constructs is introduced in §1.7.)

2.1.3.1 Declarations and Naming

2.1.3.1.1 Binding: Discussion

A TAG is represented in TDF by either a NAT or a UNIQUE value.

The following constructs and no others introduce TAGs. Each of them determines program structure.

conditional §2.1.3.5.5, EXCEPTION_HANDLER §2.2.1.1, identify §2.1.3.1.3, make_untraced_procedure §2.1.3.6.2, make_traced_procedure §2.3.3.14, repeat §2.1.3.5.6, labelled §2.1.3.5.7, variable §2.1.3.1.4, variable_no_init §2.1.3.1.5

During the evaluation of each of these constructs a value, *v*, is produced which is bound to a TAG, *t*, during the evaluation of an EXP. This means that during the evaluation of the EXP, evaluation of *obtain_tag(t)* will produce the value *v*. Only those TAGs which have been introduced in this way are available for use in a *obtain_tag* construct.

Each of the TAGs introduced in a TDF capsule will be represented by a different value, and so no scope rules are needed. Avoidance of re-use of the same identifier in separate TDF capsules is achieved by using UNIQUE values for TAGs.

2.1.3.1.2 Register: Discussion

In order to pass on the information supplied by the ANSI C *register* construct, the

register argument is supplied in the declarations *identify* §2.1.3.1.3, *variable* §2.1.3.1.4 and *variable_no_init* §2.1.3.1.5. This BOOL, if true, signifies that the name is heavily used in the controlled expression and so access to the value should be as fast as possible. In the case of variable declarations, it also implies that the variable is only used in *contents* and *assign* operations.

2.1.3.1.3 identify

register: BOOL,
visible: BOOL,
name: TAG X,
def: EXP X,
body: EXP Y

-> EXP Y

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

The value delivered by *identify* is that produced by the evaluation of *body*. Thus the SHAPE of the value delivered by *identify* is the same as the SHAPE, Y, of *body*.

register gives information about the usage of *name*. If *register* is true, *name* will not be used as a non-local of a procedure. It may also be taken as an indication that *name* is heavily used within *body*, and that allocation to a register, if possible, would be advantageous.

The TAG given for *name* will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

visible specifies whether the value bound with *name* is to be made available in the event that an exception occurs during the evaluation of *body* and the exception is diagnosed. If *visible* is true, translators shall arrange for it to be available.

In the case where *def* is simply *obtain_tag(t)*, translators should produce no code, since this usage of *identify* amounts to a mere renaming of *t* as *name*. Similarly, if *def* is constructed by a succession of *field* operations on *obtain_tag(t)*, translators should produce no code, since this usage amounts to the naming of a part of a value which has already been named.

Cross-reference: Register: Discussion §2.1.3.1.2, Binding: Discussion §2.1.3.1.1, Exceptions: Discussion §2.2.3.2.3, obtain_tag §2.1.3.1.6, field §2.1.3.9.3

2.1.3.1.4 variable

register: BOOL,
visible: BOOL,
name: 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 whose SHAPE is X. The space is initialised with *v*. Then *body* is evaluated. During the evaluation, an original POINTER pointing to the allocated space is bound to *name*. This means that inside *body* an evaluation of *obtain_tag(name)* will produce an original POINTER pointing to the space. If *variable* occurs inside an UNTRACED_PROC, then the POINTER will be an UNTRACED_POINTER. If it occurs inside a TRACED_PROC, it will be a PART_POINTER.

The value delivered by *variable* is that produced by the evaluation of *body*. Thus the SHAPE of the value delivered by *variable* is the same as the SHAPE, Y, of *body*.

register gives information about the usage of *name*. If *register* is true, *name* will not be used as a non-local of a procedure. It will be used only in *assign*, *contents* and *part_field* constructs. It may also be taken as an indication that *name* is heavily used within *body*, and that allocation to a register, if possible, would be advantageous.

The TAG used for *name* will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

visible specifies whether the space associated with *name* is to be made available in the case that an exception occurs during *body* and the exception is diagnosed. If *visible* is true, translators shall arrange for it to be available.

The POINTER associated with *name* has a lifetime limited to the execution of *body*. Any attempt to use it when *body* is not being executed is undefined.

The sharing properties of the POINTER are discussed in §2.1.3.4.1.1.

In ANSI C, a use of *obtain_tag(name)* is equivalent to a use of &. The use of *contents(obtain_tag(name))* is equivalent to the use of the right hand value of a variable, and the use of *assign(obtain_tag(name), x)* is equivalent to the use of the left hand value of a variable.

Cross-reference: Register: Discussion §2.1.3.1.2, Lifetimes: Discussion §2.1.3.12, Binding: Discussion §2.1.3.1.1, Exceptions: Discussion §2.2.3.2.3, sharing

§2.1.3.4.1.1, generate §2.3.3.1, whole_to_part §2.3.3.2, add_to_ptr §2.1.3.4.2, subtract_from_ptr §2.1.3.4.3, index §2.2.3.2.17, part_field §2.1.3.4.4, shake §2.3.3.12, firm §2.3.3.13, original POINTERs §2.1.3.4.1.3, obtain_tag §2.1.3.1.6

2.1.3.1.5 variable_no_init

register: BOOL,
visible: BOOL,
name: TAG POINTER(sh),
sh: SHAPE, {sh will be SOME-free}
body: EXP Y

-> EXP Y

Space is allocated to hold a value whose SHAPE is *sh*. The space is not initialised. *body* is then evaluated. During the evaluation, an original POINTER pointing to the allocated space is bound to *name*. This means that inside *body* an evaluation of *obtain_tag(name)* will produce an original POINTER pointing to the space. If the contents of the space are examined before a value is assigned into it, the effect is undefined. If *variable_no_init* occurs inside an UNTRACED_PROC, then the POINTER will be an UNTRACED_POINTER. If it occurs inside a TRACED_PROC, it will be a PART_POINTER.

The value delivered by *variable_no_init* is that produced by the evaluation of *body*. Thus the SHAPE of the value delivered by *variable_no_init* is the same as the SHAPE, Y, of *body*.

register gives information about the usage of *name*. If *register* is true, *name* will not be used as a non-local of a procedure. It will be used only in *assign*, *contents* and *part_field* constructs. It may also be taken as an indication that *name* is heavily used within *body*, and that allocation to a register, if possible, would be advantageous.

The TAG used for *name* will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not hapnen. See §2.1.3.1.1 for a discussion of this point.

visible specifies whether the space associated with *name* is to be made available in the case that an exception occurs during *body* and the exception is diagnosed. If *visible* is true, translators shall arrange for it to be available.

The POINTER associated with *name* has a lifetime limited to the execution of *body*. Any attempt to use it when *body* is not being executed is undefined.

The sharing properties of the POINTER are discussed in §2.1.3.4.1.1.

In ANSI C, a use of *obtain_tag(name)* is equivalent to a use of &. The use of

contents(obtain_tag(name)) is equivalent to the use of the right hand value of a variable, and the use of *assign(obtain_tag(name), x)* is equivalent to the use of the left hand value of a variable.

If *sh* is a compound SHAPE, such as a TUPLE, then the space to hold it may contain areas of padding in order to conform to an architcture's alignment rules. Though equality between TUPLEs depends solely on the equality of the components, installers may very well implement this operation by comparing the whole of the TUPLEs, including any padding. Translators therefore need to make sure that such padding has a standard vaue. This means that some pseudo-initialisation operation on the space for this variable may be necessary, even for pure ANSI C translators.

Cross-reference: Register: Discussion §2.1.3.1.2, Lifetimes: Discussion §2.1.3.12, Binding: Discussion §2.1.3.1.1, Exceptions: Discussion §2.2.3.2.3, sharing §2.1.3.4.1.1, generate §2.3.3.1, whole_to_part §2.3.3.2, add_to_ptr §2.1.3.4.2, subtract_from_ptr §2.1.3.4.3, index §2.2.3.2.17, part_field §2.1.3.4.4, shake §2.3.3.12, firm §2.3.3.13, obtain_tag §2.1.3.1.6

2.1.3.1.6 obtain_tag

name: TAG X

-> EXP X

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

Cross-reference: Binding: Discussion §2.1.3.1.1

2.1.3.2 Integers

2.1.3.2.1 Character Sets: Discussion

TDF, as a representation of program, does not manipulate characters explicitly. Instead, they are represented by integers. Conventions for mapping characters onto integers are required.

Characters appear in programs, and need to correspond to the characters which appear on the printers and displays of target machines. But the hardware of target machines can use a multiplicity of different collating sequences for characters. In order to achieve portability of TDF programs it is necessary to choose a standard representation for characters in the TDF itself. Translation to the collating sequence

for the hardware devices then should occur only on the point of transmission to those devices. The standard is separate from the definition of TDF.

Since ANSI C is compatible with ASCII and Ada makes it mandatory, TDF standardises on ASCII.

Other character sets, such as Japanese, may need to be represented as strings written in programs. But not all target machines have Japanese printers. To conform with the need for portability of TDF programs a similar standard represention of characters in TDF and translation at the device will be needed, for those programs and target machines which use Japanese characters. Multi-byte characters will probably be used. Similar standards are needed for all such character sets. These will have to be standardised as the need arises.

The customisation of user's programs to give messages in the user's own language can be achieved by tokenising the messages (or the collection of messages) and making the substitutions during installation of the program.

Cross-reference: tokenisation §1.5.3, TDF: Scenario of Use §1.1

2.1.3.2.2 make_int

v: VARIETY,
value: SIGNED_NAT

-> EXP INTEGER(v)

An integer value is delivered whose value is given by *value*, and whose VARIETY is given by *v*. The integer value *value* will lie between the bounds of *v*. This ensures that *value* is representable as an integer of VARIETY *v*.

Cross-reference: integer SHAPEs §2.1.2.3.1, SIGNED_NAT §2.1.1.4

2.1.3.2.3 plus

ov_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The sum of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *V*, is unsigned, the operation is performed modulo 2^bits(*V*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1

2.1.3.2.4 minus

ov_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The difference of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *V*, is unsigned, the operation is performed modulo 2^bits(*V*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1

2.1.3.2.5 mult

ov_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The product of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *V*, is unsigned, the operation is performed modulo 2^bits(*V*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Translators should if possible optimise multiplication by powers of 2 and any relevant constants.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1

2.1.3.2.6 Kinds of Division: Discussion

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 D1 q = n

where p = n*q + (p M1 q)
sign(p M1 q) = sign(q)
0 ≤ |p M1 q| < |q|

Class 2:

p D2 q = n

where p = n*q + (p M2 q)
sign(p M2 q) = sign(p)
0 ≤ |p M2 q| < |q|

2.1.3.2.7 div1

ov_err:ERROR_TREATMENT,
div0_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* D1 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Translators should if possible optimise division by constants, especially powers of 2.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, Kinds of Division: Discussion(for D1) §2.1.3.2.6

2.1.3.2.8 div2

ov_err:ERROR_TREATMENT,
div0_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* D2 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Translators should if possible optimise division by constants, especially powers of 2. This is possible if *V* is unsigned.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, Kinds of Division: Discussion(for D2) §2.1.3.2.6

2.1.3.2.9 mod

div0_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* M1 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Translators should if possible optimise modulus by powers of 2.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1,
Kinds of Division: Discussion(for M1) §2.1.3.2.6

2.1.3.2.10 rem2

div0_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* M2 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1,
Kinds of Division: Discussion(for M2) §2.1.3.2.6

2.1.3.2.11 exact_divide

arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY.

The quotient of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments. *b* will be an exact divisor of *a*.

Cross-reference: integer SHAPEs §2.1.2.3.1

2.1.3.2.12 negate

ov_err:ERROR_TREATMENT,
arg:EXP INTEGER(V)

-> EXP INTEGER(V)

arg is evaluated to produce an integer value, *a*. The negation of *a* is delivered as the result of the construct, which has the same SHAPE as the construct's argument.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

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

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1

2.1.3.2.13 abs

ov_err:ERROR_TREATMENT,
arg:EXP INTEGER(V)

-> EXP INTEGER(V)

arg is evaluated to produce an integer value, *a*. The absolute value of *a* is delivered as the result of the construct, which has the same SHAPE as the construct's argument.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

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

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1

2.1.3.2.14 Number Conversion: Discussion

There is no automatic conversion between integer VARIETYs.

Conversions between integer VARIETYs are carried out by *change_var*. In every case, if the same integer is expressible in the destination VARIETY, this integer expressed in the destination VARIETY is the result.

Certain other conversions are provided which are easy to implement in 2's complement machines, and possible in other representations.

When a negative signed integer is converted to an unsigned limited VARIETY whose *maxint* is greater than both the modulus of the *minint* and the *maxint* of the source VARIETY, the resulting value is obtained by adding one more than the *maxint* of the target VARIETY.

When an integer is converted to an unsigned VARIETY with *maxint* less than either the modulus of the *minint* or the *maxint* of the source VARIETY, the result is the remained (M1) on division by the number one greater than the *maxint* of the target VARIETY.

All other conversions are target-defined.

Cross-reference: Kinds of Division: Discussion(for M1) §2.1.3.2.6, change_var §2.1.3.2.15

2.1.3.2.15 change_var

w:VARIETY,
arg:EXP INTEGER(V)

-> EXP INTEGER(w)

arg is evaluated to produce an integer value, *a*. If *a* is expressible in VARIETY *w*, then it is delivered as the result of the construct. The result has the SHAPE INTEGER(*w*).

Certain other special target-dependent conversions are defined in §2.1.3.2.14. No other conversions are defined.

Cross-reference: Number Conversion: Discussion §2.1.3.2.14, integer SHAPEs §2.1.2.3.1

2.1.3.2.16 maxint

v:VARIETY

-> EXP INTEGER(v)

An integer value is created and delivered which is the maximum integer expressible in the VARIETY, v. Note that this is not necessarily the same as the largest integer given in the VARIETY definition. For example, a translator might well represent the VARIETY (5,8) in one byte. In this case, then, the value delivered by maxint((5,8)) would be 255.

The value delivered by *maxint* is target-defined.

Cross-reference: integer SHAPEs §2.1.2.3.1

2.1.3.2.17 minint

v:VARIETY

-> EXP INTEGER(v)

An integer value is created and delivered which is the minimum integer expressible in the VARIETY, v. Note that this is not necessarily the same as the largest integer given in the VARIETY definition. For example, a translator might well represent the VARIETY (5,8) in one byte. In this case, then, the value delivered by maxint((5,8)) would be 0.

The value delivered by *minint* is target-defined.

Cross-reference: integer SHAPEs §2.1.2.3.1

2.1.3.2.18 shift_left

ov_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V1),
arg2:EXP INTEGER(V2)

-> EXP INTEGER(V1)

arg1 and *arg2* are evaluated to produce values *a* and *places*. The result is equivalent to:

if places < 0
then divl(ov_err, impossible, a, $2^{-places}$*)*
else mult(ov_err, a, 2^{places}*)*

The implementation is expected to optimise cases where the number of shifts is a constant.

Cross-reference: divl §2.1.3.2.7, mult §2.1.3.2.5

2.1.3.2.19 shift_right

ov_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V1),
arg2:EXP INTEGER(V2)

-> EXP INTEGER(V1)

arg1 and *arg2* are evaluated to produce values *a* and *places*. The result is equivalent to:

if places > 0
then div1(ov_err, impossible, a, 2^{places}*)*
else mult(ov_err, a, $2^{-places}$*)*

The implementation is expected to optimise the cases where the number of shifts is a constant.

Cross-reference: div1 §2.1.3.2.7, mult §2.1.3.2.5

2.1.3.2.20 round

ov_err:ERROR_TREATMENT,
v:VARIETY,
arg:EXP FLOAT(F)

-> EXP INTEGER(v)

arg is evaluated to produce a floating point value, *a*. If the nearest integer to *a* is expressible in VARIETY *v*, then a value of that integer is created and delivered.

However, if that nearest integer cannot be expressed in VARIETY *v*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *v*, is unsigned, the operation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, floating point SHAPEs §2.1.2.3.2

2.1.3.2.21 truncate

ov_err:ERROR_TREATMENT,
v:VARIETY,
arg:EXP FLOAT(F)

-> EXP INTEGER(v)

arg is evaluated to produce a floating point value, *a*. If the integer part of *a* is expressible in VARIETY *v*, then a value of that integer is created and delivered.

However, if that nearest integer cannot be expressed in VARIETY *v*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *v*, is unsigned, the operation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, floating point SHAPEs §2.1.2.3.2

2.1.3.2.22 bits_to_integer

v:VARIETY,
ov_err: ERROR_TREATMENT,
arg:EXP NOF(BIT, N)

-> EXP INTEGER(v)

arg is evaluated to produce an NOF(BIT, N) value, *r*. This value is converted to an integer, *a*, of VARIETY *v*, which is delivered.

The manner in which *a* is calculated depends on the VARIETY *v*. If *v* is an unsigned VARIETY, then *a* is derived as follows:

$$a = \text{If } q \le \text{maxint}(v) \text{ Then } q \text{ Else } q - \text{maxint}(v) - 1$$

where:

$$q = \Sigma_{i=0}^{N} r_i * 2^i$$

In this case *a* is always non-negative.

However, if *v* is a signed VARIETY, then *a* is derived as follows:

$$a = (\text{If } r_N = 1 \text{ Then } -1 \text{ Else } 1) * \begin{array}{l}\text{If } q \le \text{maxint}(v) \\ \quad \text{Then } q \\ \quad \text{Else } q - \text{maxint}(v) - 1\end{array}$$

where:

$$q = \Sigma_{i=0}^{N-1} r_i * 2^i$$

In this case *a* may be negative.

If the result cannot be expressed in the required VARIETY, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *v*, is unsigned, the operation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.1.3.2.23 div_rem1

ov_err:ERROR_TREATMENT,
div0_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP TUPLE(INTEGER(V), INTEGER(V))

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. A TUPLE of (*a* D1 *b*, *a* M1 *b*) is delivered as the result.

If the result cannot be expressed in the VARIETY *V*, an overflow error is caused and handled in the manner specified by *ov_err*. This only occurs for signed VARIETYs in the special case of dividing *minint* by -1.

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, Kinds of Division: Discussion(for D1 and M1) §2.1.3.2.6

2.1.3.2.24 div_rem2

ov_err:ERROR_TREATMENT,
div0_err:ERROR_TREATMENT,
arg1:EXP INTEGER(V),
arg2:EXP INTEGER(V)

-> EXP TUPLE(INTEGER(V), INTEGER(V))

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. A TUPLE of (*a* D2 *b*, *a* M2 *b*) is delivered as the result.

If the result cannot be expressed in the VARIETY *V*, an overflow error is caused and handled in the manner specified by *ov_err*. This only occurs for signed varieties in the special case of dividing *minint* by -1.

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1,
Kinds of Division: Discussion(for D2 and M2) §2.1.3.2.6

2.1.3.2.25 integer_test

ntest:NTEST,
bad:LABEL,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP TOP

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same integer VARIETY. These values are compared using the test *ntest*. If the test succeeds, the construct delivers a value of SHAPE *top*. If it fails, control passes to the LABEL *bad* with a value of SHAPE *top*. Since the only way in which the construct can deliver a result is when the test succeeds, the SHAPE of the result of the construct is itself *top*.

To give an example, if *ntest* is *greater*, then if *a* is greater than *b* the construct delivers a value of SHAPE *top*. If *a* is not greater than *b* is false, control passes to the LABEL *bad*.

Cross-reference: LABEL §2.1.1.10, TOP §2.1.2.1.2, NTEST §2.1.1.11, integer SHAPEs §2.1.2.3.1

2.1.3.2.26 bit_integer_test

ntest:NTEST,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP BIT

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same integer VARIETY. These values are compared using the test *ntest*. If the test succeeds, a *true* BIT is delivered. Otherwise, a *false* BIT is delivered.

Cross-reference: NTEST §2.1.1.11, integer SHAPEs §2.1.2.3.1

2.1.3.2.27 integer_to_bits

arg: EXP INTEGER(V)

-> EXP NOF(BIT, n)

arg1 is evaluated to produce an integer value *a*. A value *r* of SHAPE NOF(BIT, *n*) is created and delivered, where *n* shall be the smallest number of bits required to represent the full (ie. *minint* to *maxint*) range of values in INTEGER(V).

The value *r* is chosen so that if *a* is non-negative

$$a = \Sigma_{i=0}^{n-1} r_i * 2^i$$

.. and if *a* is negative

$$a = \Sigma_{i=0}^{n-1} r_i * 2^i - \text{maxint}(V) - 1$$

On twos-complement machines, translators should not need to generate any code to implement this operation.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, integer SHAPEs §2.1.2.3.1

2.1.3.3 Floating Point Values

These operations are defined for all FLOATING_VARIETYs.

It will be desirable to add floating point operations such as *sqrt* and *sine*. The standardisation and introduction of these operations (by tokenising) is deferred.

2.1.3.3.1 make_floating

f: FLOATING_VARIETY,
mantissa: SIGNED_NAT,
base: NAT,
exponent: SIGNED_NAT

-> EXP FLOAT(f)

A floating point value *v* of FLOATING_VARIETY *f* is created and delivered. The value is the nearest to

$$\text{mantissa} \times (\text{base}^{\text{exponent}})$$

v will be representable in the FLOATING_VARIETY *f*.

Cross-reference: floating point SHAPEs §2.1.2.3.2, NAT §2.1.1.3, SIGNED_NAT §2.1.1.4

2.1.3.3.2 floating_plus

ov_err:ERROR_TREATMENT,
arg1:EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The sum of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2

2.1.3.3.3 floating_minus

ov_err:ERROR_TREATMENT,
arg1:EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The difference of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2

2.1.3.3.4 floating_mult

ov_err:ERROR_TREATMENT,
arg1:EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The product of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2

2.1.3.3.5 floating_div

ov_err:ERROR_TREATMENT,
div0_err:ERROR_TREATMENT,
arg1:EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The quotient of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

If *b* is zero a divide-by-zero error is produced and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2

2.1.3.3.6 floating_rem

div0_err:ERROR_TREATMENT,
arg1:EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. *a* is divided by *b* and the remainder delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If *b* is zero a divide-by-zero error is produced and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2

2.1.3.3.7 floating_negate

ov_err:ERROR_TREATMENT,
arg:EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 is evaluated to produce a floating point value, *a*. The negation of *a* is delivered as the result of the construct, which has the same SHAPE as the construct's argument.

If the result cannot be expressed in the FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2

2.1.3.3.8 float

ov_err: ERROR_HANDLER,
f:FLOATING_VARIETY,
arg:EXP INTEGER(V)

-> EXP FLOAT(f)

arg is evaluated to produce an integer value, *a*. An equal floating point value of FLOATING_VARIETY *f* is created and delivered. Any rounding necessary is target-defined.

If the integer value *a* is not representable in FLOATING_VARIETY *f* an overflow error is generated and handled by *ov_err*. If *ov_err* is *ignore* the effect is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, floating point SHAPEs §2.1.2.3.2, integer SHAPEs §2.1.2.3.1

2.1.3.3.9 change_floating_variety

ov_err:ERROR_TREATMENT,
f:FLOATING_VARIETY,
arg:EXP FLOAT(F)

-> EXP FLOAT(f)

arg is evaluated to produce a floating point value, *a*. A floating point value is created

and delivered which has FLOATING_VARIETY *f* and is equal to *a*. This conversion is target-defined.

If *a* cannot be expressed in FLOATING_VARIETY *f*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

Cross-reference: floating point SHAPEs §2.1.2.3.2

2.1.3.3.10 floating_test

ntest:NTEST,
bad:LABEL,
arg1: EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP TOP

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. These values are compared using the test *ntest*. If the test succeeds the construct delivers a value of SHAPE *top*. If it fails, control passes to the LABEL *bad* with a value of SHAPE *top*. Since the only way in which the construct can deliver a result is when the test succeeds, the SHAPE of the result of the construct is itself *top*.

To give an example, if *ntest* is *greater*, then if *a* is greater than *b* the construct delivers a value of SHAPE *top*. If *a* is not greater than *b* is false, control passes to the LABEL *bad*.

Cross-reference: LABEL §2.1.1.10, TOP §2.1.2.1.2, NTEST §2.1.1.11, floating point SHAPEs §2.1.2.3.2

2.1.3.3.11 bit_floating_test

ntest:NTEST,
arg1: EXP FLOAT(F),
arg2:EXP FLOAT(F)

-> EXP BIT

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. These values are compared using the test *ntest*. If the test succeeds a *true* BIT is delivered. Otherwise, a *false* BIT is delivered.

Cross-reference: NTEST §2.1.1.11, floating point SHAPEs §2.1.2.3.2

2.1.3.4 POINTERs

2.1.3.4.1 POINTERs: Discussion

Before describing the Level 0 constructs which create and manipulate POINTERs, it is useful to introduce four important concepts - sharing, null POINTERs, original POINTERs and proper POINTERs.

2.1.3.4.1.1 Sharing

Sharing is a concept which relates only to POINTERs. If a POINTER, *a*, points to a space, *a_space*, and a POINTER, *b*, points to a space, *b_space*, and *a_space* and *b_space* overlap, then *a* and *b* are said to share. In other words, if an assignment operation (§2.1.3.4.5) to *b* can change the result of using a contents operation (§2.1.3.4.6) on *a*, or vice versa, then *a* and *b* share.

All of the definitions of operations which produce WHOLE_POINTERs, PART_POINTERs, SHAKY_POINTERs or variables define the sharing properties of the POINTERs they create.

Null POINTERs cannot share.

Cross-reference: assign §2.1.3.4.5, contents §2.1.3.4.6, WHOLE_POINTER SHAPEs §2.3.2.1.2, PART_POINTER SHAPEs §2.3.2.1.4, null POINTERs §2.1.3.4.1.2

2.1.3.4.1.2 Null POINTERs

Null POINTERs are required in order to provide a suitable value to put at the end of a list and for similar puposes. Any attempt to obtain the contents of a null POINTER, or to use it as the destination in an assign operation, is defined to produce a detectable error.

There is just one null WHOLE_POINTER value, the value produced by *make_null_whole_pointer*. Null WHOLE_POINTERs will therefore have equal representations.

By contrast, there are many possible null PART_POINTER values. *make_null_part_pointer* will always produce the same null PART_POINTER value. The following equation illustrates this:

whole_to_part(make_null_whole_pointer)
= *make_null_part_pointer*

However, *part_field* applied to a null POINTER offers the possibility of producing many different null PART_POINTER values.

If *index*, *add_to_ptr* or *subtract_from_ptr* are applied to a null POINTER the effect is undefined.

Cross-reference: add_to_ptr §2.1.3.4.2, assign §2.1.3.4.5, contents §2.1.3.4.6, firm §2.3.3.13, index §2.2.3.2.17, make_null_part_pointer §2.3.3.4, make_null_whole_pointer §2.3.3.3, part_field §2.1.3.4.4, shake §2.3.3.12, subtract_from_ptr §2.1.3.4.3

2.1.3.4.1.3 Original POINTERs

A WHOLE_POINTER (produced by *generate*) or a PART_POINTER identified in a *variable* or *variable_no_init* declaration are original POINTERs. Original POINTERs are only equal if they are copies of a value produced by one execution of *generate*, or one execution of a variable declaration.

Every POINTER is said to be derived from an original POINTER if and only if it is either a copy of that POINTER or obtained from it by a succession of the following operations:-

add_to_ptr §2.1.3.4.2, firm §2.3.3.13, index §2.2.3.2.17, part_field §2.1.3.4.4, shake §2.3.3.12, subtract_from_ptr §2.1.3.4.3, whole_to_part §2.3.3.2

Every POINTER is derived from just one original POINTER.

2.1.3.4.1.4 Proper POINTERs

A proper POINTER is a POINTER which points to a space equal to or contained within the space to which its parent original POINTER pointed. Thus every original POINTER is a proper POINTER. A PART_POINTER may or may not be a proper POINTER. Both proper and improper POINTERs are legal and defined in TDF, but an attempt to take the contents of an improper POINTER is undefined, as is the result of an attempt to assign a value to an improper POINTER.

Cross-reference: original POINTERs §2.1.3.4.1.3, contents §2.1.3.4.6, assign §2.1.3.4.5

2.1.3.4.2 add_to_ptr

ptr: EXP POINTER(X),
off: EXP OFFSET(X,Y)

-> EXP PART_POINTER(Y)

ptr is evaluated to produce a POINTER *p* and *off* to produce an OFFSET value *o*. A PART_POINTER is created and delivered which points to space for a value of SHAPE Y offset ahead by *o* from the space pointed to by *p*. If *p* is null, the result is undefined.

The result may share with *p*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, SHAPEs SIZEs and OFFSETs §1.5.1, sharing §2.1.3.4.1.1, Lifetimes: Discussion §2.1.3.12

2.1.3.4.3 subtract_from_ptr

ptr: EXP POINTER(X),
off: EXP OFFSET(W,X)

-> EXP PART_POINTER(W)

ptr is evaluated to produce a POINTER *p* and *off* to produce an OFFSET value *o*. A PART_POINTER is created and delivered which points to space for a value of SHAPE W offset back by *o* from the space pointed to by *p*. If *p* is null, the result is undefined.

The result may share with *p*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, SHAPEs SIZEs and OFFSETs §1.5.1, sharing §2.1.3.4.1.1, Lifetimes: Discussion §2.1.3.12

2.1.3.4.4 part_field

component: NAT,
arg_shape: TUPLE $\Pi_{i=1}^{n} S_i$, {n ≥ 1} {1 <= component <= n}
{the S_i 1 <= i < component are SOME-free}
ptr: EXP UNTRACED_POINTER(X)

-> EXP UNTRACED_POINTER(Y) {Y = $S_{component}$}

ptr is evaluated to produce a POINTER *p* to a space, *sp*, containing a TUPLE of SHAPE *arg_shape*. A PART_POINTER(Y) pointing to that space within *sp* which contains the *component*-th field of the TUPLE is created and delivered. The result and *p* share.

(In Level 2, a WHOLE_POINTER(X) argument may be supplied, in which case the SHAPE of the result is PART_POINTER(X).)

If *p* is a null POINTER, then so is the result. However, they need not be equal null POINTERs.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, sharing §2.1.3.4.1.1, Lifetimes: Discussion §2.1.3.12, tuple §1.5.1.1

2.1.3.4.5 assign

```
err: ERROR_TREATMENT,
ptr: EXP POINTER(X),
val: EXP Y

  -> EXP TOP
```

ptr and *val* are evaluated to produce values, *p* and *v*. The POINTER, *p*, will not be volatile in the sense of ANSI C. The value v is put into the space pointed to by *p*. If *p* is a null POINTER then a *null_pointer* error occurs which is handled as specified by *err*. If *err* is *ignore* its effect is undefined.

If *X* is not the same as *Y*, the effect is undefined. They will always be the same in TDF derived from strictly typed languages such as Ada, but may not be in TDF derived from ANSI C.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, ERROR_TREATMENT §2.1.1.12, Lifetimes: Discussion §2.1.3.12, proper POINTERs §2.1.3.4.1.4

2.1.3.4.6 contents

```
is_null:ERROR_TREATMENT,
sh: SHAPE,                  {sh will be SOME-free}
pointer: EXP POINTER(X)

  -> EXP sh
```

pointer is evaluated to produce a value *p*. The POINTER *p* will not be volatile in the sense of ANSI C. The content of the space pointed to by *p* is delivered as the result. If *p* is a null POINTER, then a *null_pointer* error is caused and handled according to *is_null*. If *is_null* is *ignore*, the effect is undefined.

If *sh* is not the same as *X*, the effect is undefined. They will always be the same in TDF derived from strictly typed languages such as Ada, but may not be in TDF derived from ANSI C.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, ERROR_TREATMENT §2.1.1.12, proper POINTERs §2.1.3.4.1.4

2.1.3.4.7 assign_to_volatile

err: ERROR_TREATMENT,
ptr: EXP POINTER(X),
val: EXP Y

-> EXP TOP

ptr and *val* are evaluated to produce values, *p* and *v*. The POINTER *p*, will be volatile in the sense of ANSI C. The value v is put into the space pointed to by p. If *p* is a null POINTER then a *null_pointer* error occurs which is handled as specified by *err*. If *err* is *ignore* its effect is undefined.

If *X* is not the same as *Y*, the effect is undefined. They will always be the same in TDF derived from strictly typed languages such as Ada, but may not be in TDF derived from ANSI C.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, ERROR_TREATMENT §2.1.1.12, Lifetimes: Discussion §2.1.3.12, proper POINTERs §2.1.3.4.1.4

2.1.3.4.8 contents_of_volatile

```
is_null:ERROR_TREATMENT,
sh: SHAPE,                {sh will be SOME-free}
pointer: EXP POINTER(X)

  -> EXP sh
```

pointer is evaluated to produce a value *p*. The POINTER *p* will be volatile in the sense of ANSI C. The content of the space pointed to by *p* is delivered as the result. If *p* is a null POINTER, then a *null_pointer* error is caused and handled according to *is_null*. If *is_null* is *ignore*, the effect is undefined.

If *sh* is not the same as *X*, the effect is undefined. They will always be the same in TDF derived from strictly typed languages such as Ada, but may not be in TDF derived from ANSI C.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, ERROR_TREATMENT §2.1.1.12, proper POINTERs §2.1.3.4.1.4

2.1.3.4.9 move_contents

```
no_overlap:BOOL,
arg1: EXP POINTER(X),
arg2: EXP POINTER(Y),
arg3: EXP SIZE(Z)      {Z need not be SOME-free}

  -> EXP TOP
```

arg1, *arg2* and *arg3* are evaluated to produce values, *a*, *b* and *c*. The amount of data specified by *c* is moved from the space pointed to by *a* to that pointed to by *b*.

no_overlap controls the behaviour in the case that the source and destination spaces overlap. If it is false, the move will be performed in such a way that the resulting state of *b* is the same as if the spaces had not overlapped.

If *no_overlap* is true, the source and destination spaces will not overlap and translators can optimise the code which they produce for this construct accordingly.

If *X* is not the same as *Y*, the effect is undefined. They will always be the same in TDF derived from strictly typed languages such as Ada.

If the space to which *b* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

Cross-reference: Pointers: Discussion §2.1.3.4.1, SHAPEs SIZEs and OFFSETs §1.5.1, Lifetimes: Discussion §2.1.3.12, proper POINTERs §2.1.3.4.1.4

2.1.3.4.10 assign_bits

ptr_arg: EXP POINTER(X),
bit_offset: EXP OFFSET(NOF(BIT,N),Y),
nbits: EXP NOF(BIT, N)

-> EXP TOP

ptr_arg, *bit_offset* and *nbits* are evaluated to produce values *p*, *b* and *n*. The NOF value, *n*, is assigned into the space pointed to by *p* starting at the OFFSET *b*.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

Note that there is no requirement that the POINTER's SHAPE be an NOF(BIT, ..).

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.1.3.4.11 contents_bits

number:NAT
ptr: EXP POINTER(X),
bit_offset: EXP OFFSET(NOF(BIT,N),Y)

-> EXP NOF(BIT, number)

ptr and *bit_offset* are evaluated to produce values *p* and *off*. The contents of the space pointed to by *p*, starting at the OFFSET *off*, are delivered as a value of SHAPE NOF(BIT,number).

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.1.3.4.12 pointer_test

test: NTEST,
bad: LABEL,
arg1: EXP POINTER(X),
arg2: EXP POINTER(Y)

-> EXP TOP

arg1 and *arg2* are evaluated to produce POINTER values, *a* and *b*. These values are compared using the test specified by *test*. If the test succeeds, the construct delivers *top*. If the test fails, control passes to the LABEL *bad* with *top*. Since the only way in which *pointer_test* can deliver a result is when the test succeeds, the SHAPE of the result of *pointer_test* is itself *top*.

Unless X and Y are the same and *a* and *b* are derived from the same original POINTER, the effect is implementation defined.

Cross-reference: LABEL §2.1.1.10, TOP §2.1.2.1.2, NTEST §2.1.1.11, SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.4.13 bit_pointer_test

test: NTEST,
arg1: EXP POINTER(X),
arg2:EXP POINTER(Y)

-> EXP BIT

arg1 and *arg2* are evaluated to produce POINTER values, *a* and *b*. These values are compared using the test specified by *test*. If the test succeeds, a *true* BIT is delivered. Otherwise, a *false* BIT is delivered.

Unless X and Y are the same and *a* and *b* are derived from the same original POINTER, the effect is implementation defined.

Cross-reference: NTEST §2.1.1.11, SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.4.14 subtract_pointers

arg1: EXP POINTER(X),
arg2: EXP POINTER(Y)

-> EXP OFFSET(Y,X)

arg1 and *arg2* are evaluated to produce POINTER values, *a* and *b*. If *a* and *b* are derived from the same original POINTER, then the OFFSET of *a* from *b* is delivered as the result.

If *a* and *b* are not derived from the same original POINTER, the effect is undefined.

Cross-reference: Pointers: Discussion §2.1.3.4.1, original POINTERs §2.1.3.4.1.3, SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.4.15 ptr_is_null

not_null:LABEL,
arg: EXP POINTER(X)

-> EXP TOP

arg is evaluated to produce a POINTER value, *v*. If *v* is found to be a null POINTER, the construct delivers a value of SHAPE *top*. If it is not a null POINTER, control passes to the LABEL *not_null* with *top*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, LABEL §2.1.1.10

2.1.3.4.16 ptr_not_null

is_null:LABEL,
arg: EXP POINTER(X)

-> EXP TOP

arg is evaluated to produce a POINTER value, *v*. If *v* is found not to be a null POINTER the construct delivers a value of SHAPE *top*. If it is a null POINTER, control passes to the LABEL *is_null* with *top*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, LABEL §2.1.1.10

2.1.3.5 Program Structure and Flow of Control

2.1.3.5.1 Availability of LABELs: Discussion

Only those LABELs which are available can be used in constructs which can cause control to go to a LABEL. These constructs are:-

assign §2.1.3.4.5, bits_to_integer §2.1.3.2.22, case §2.1.3.5.4,

change_floating_variety §2.1.3.3.9, contents §2.1.3.4.6, div1 §2.1.3.2.7, div2 §2.1.3.2.8, equal_contents §2.2.3.2.1, firm §2.3.3.13, floating_div §2.1.3.3.5, floating_minus §2.1.3.3.3, floating_mult §2.1.3.3.4, floating_negate §2.1.3.3.7, floating_plus §2.1.3.3.2, floating_rem §2.1.3.3.6, floating_test §2.1.3.3.10, floor §2.3.3.9, goto §2.1.3.5.8, integer_test §2.1.3.2.25, minus §2.1.3.2.4, mod §2.1.3.2.9, div_rem1 §2.1.3.2.23, div_rem2 §2.1.3.2.24, mult §2.1.3.2.5, negate §2.1.3.2.12, not_equal_contents §2.2.3.2.2, plus §2.1.3.2.3, ptr_not_null §2.1.3.4.16, rem2 §2.1.3.2.10, round §2.1.3.2.20, shift_left §2.1.3.2.18, shift_right §2.1.3.2.19, pointer_test §2.1.3.4.12, test_eq §2.1.3.10.2, test_neq §2.1.3.10.4, truncate §2.1.3.2.21

Labels are made available in components of certain control structure constructs. They are available only in the places specified in the descriptions of these constructs. These constructs are:-

conditional §2.1.3.5.5, repeat §2.1.3.5.6, labelled §2.1.3.5.7

The LABELs which are available at the point of a *make_untraced_procedure* or *make_traced_procedure* construct are also available in the body of the procedure. The use, inside the body of a procedure, of LABELs introduced outside it, will limit the lifetime of the procedure. (The concept of lifetime is introduced in §2.1.3.12.)

conditional, *repeat* and *labelled* may contain procedures which use the LABELs as non-locals (for caveats on this, see §2.1.3.6). In these cases, a change of control flow using *goto*, for example, would almost certainly have to be translated to change more than just the program counter - for instance, the stack-frame current at the introduction of the LABEL might have to be reset.

The other construct in which a LABEL can be used is *make_label_value* (§2.1.3.5.11). This gives a LABEL_VALUE which can subsequently be used as the parameter of *jump* (§2.1.3.5.9) to cause control to go to the LABEL - this is used to implement the C long jump, for example. For it to be meaningful, the construct which made the LABEL available must still be being evaluated at the evaluation of the jump (see lifetimes §2.1.3.12).

Since a LABEL_VALUE can potentially be used at any procedure level, its representation must include information defining the scope of the LABEL - at the very least, the stack-frame of the introduction of the LABEL. At a jump to the LABEL_VALUE, this scope must still be being evaluated so that it can be re-instated. A translator writer may want to ensure that this is so by doing a dynamic test before the jump.

Cross-reference: Lifetimes: Discussion §2.1.3.12, make_untraced_procedure §2.1.3.6.2, make_traced_procedure §2.3.3.14

2.1.3.5.2 Jumping with Values: Discussion

In TDF, when control passes from a *goto* or other construct to a LABEL, a value may be transferred as well, and be bound to a TAG introduced at the same place as the LABEL. This value will often be *top*, which means that nothing is being transferred, but sometimes a useful value is involved.

This style of jumping is perfectly natural in computers, although as a matter of fact few programming languages permit values to be transferred in this way. TDF provides the facility for two reasons: firstly to allow for its introduction in future systems and languages; and secondly to provide for optimisation of looping constructs. For example, the *while*, *for* etc. constructs of most programming languages have to achieve their effects by side-effecting variables declared elsewhere. There may be cases where this approach can be optimised to jumping with a value in TDF.

In an ANSI C program, *goto* will usually be with *top*, and it may well be worthwhile tokenising operations to perform such jumps.

In ANSI C programs, long jumps occur with values; the C long jump is implemented in TDF using LABEL_VALUEs.

2.1.3.5.3 sequence

statements: $\Pi_{i=1}^{n}$ EXP Y_i, $\{n > 0\}$
result: EXP X

-> EXP X

The EXPs in *statements* are evaluated in order. Then *result* is evaluated. The value delivered by *sequence* is the value produced by *result*. Thus the SHAPE of the value delivered by *sequence* is the same as the SHAPE of the value produced by *result*.

2.1.3.5.4 case

control: EXP INTEGER(V),
branches:

$\Pi_{i=1}^{n}$(lower$_i$:SIGNED_NAT, upper$_i$: SIGNED_NAT, branch$_i$:LABEL)
$\{n > 0\}$

-> EXP TOP

control is evaluated to produce an integer value, *c*. Then *c* is tested to see whether it lies inclusively between each of the *lower*$_i$ and *upper*$_i$, in order. If and when one of these tests succeeds, control immediately passes to the LABEL *branch*$_i$ with a value of SHAPE *top*. If *c* lies between none of the pairs of SIGNED_NATs, the construct delivers *top*. Since this is the only way in which *case* can deliver a result, the SHAPE of the result of *case* is itself *top*.

The sets of SIGNED_NATs will be disjoint.

Designers of translators should consider when this operation is best implemented by means of a switch jump and when by means of a succession of tests. In particular, the special case where there is only one branch should be optimised - it may be possible to use a compare against bounds instruction; as well as the case of one branch where the SIGNED_NATs are equal - which could be implemented as a simple comparison.

Cross-reference: LABEL §2.1.1.10, TOP §2.1.2.1.2

2.1.3.5.5 conditional

sh: SHAPE,
tk: (TAG sh)_OPTION,
first: EXP X,
alt_label: LABEL,
alt: EXP Y

-> EXP (X LUB Y)

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. However, if a *goto(alt_lab,exp)* is encountered during the evaluation of *first*, then evaluation of *first* will stop, *alt* will be evaluated and its result, *a*, delivered as the result of the whole construct.

Depending on the run-time behaviour of *first*, the result of the construct may be provided by *first* or by *alt*. The SHAPE of the result, which is determined at translate-time, is therefore the LUB of the SHAPES of *first* and *alt*.

During the evaluation of *alt* the value, *e*, produced by *exp* is bound to *tk*. This means that inside *alt* an evaluation of *obtain_tag(tk)* will produce the value *e*, with SHAPE *sh*. The presence of a TAG *tk* is optional. If a TAG is not supplied, then no binding occurs, and the value, *e*, is "invisible" inside *alt*.

The TAG used for *tk* will not be re-used. No rules for the effect of the hiding of one tag by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

If *alt_lab* is not used in *first*, translators should suppress the translation of *alt*, since it could never be evaluated.

Note that *alt_lab* is not available in *alt*. In consequence this operation cannot be used to provide a loop.

Cross-reference: LABEL §2.1.1.10, Availability of Labels: Discussion §2.1.3.5.1, goto §2.1.3.5.8, Exceptions: Discussion §2.2.3.2.3, Binding: Discussion §2.1.3.1.1, Jumping with Values: Discussion §2.1.3.5.2

2.1.3.5.6 repeat

tk: (TAG X)_OPTION,
start: EXP X,
repeat_label: LABEL,
body: EXP Y

-> EXP Y

start is evaluated to produce a value *st* of SHAPE X. Then *body* is evaluated. During this evaluation of *body*, *st* is bound to *tk*. This means that inside *body* an evaluation of *obtain_tag(tk)*will produce the value *st*.

If *body* produces a result, *b*, this is delivered as the result of the whole construct. However, if a *goto(repeat_label,exp)* is encountered during the evaluation of *body*, then the evaluation of *body* stops. *body* is then evaluated afresh.

During this new evaluation, the value, *e*, produced by *exp* is bound to *tk*. If a TAG is not supplied, then no binding occurs, and the value, *e*, is "invisible" inside *body*.

The looping behaviour may be repeated indefinitely.

The TAG used for *tk* will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

Cross-reference: LABEL §2.1.1.10, Availability of Labels: Discussion §2.1.3.5.1, goto §2.1.3.5.8, Exceptions: Discussion §2.2.3.2.3, Binding: Discussion §2.1.3.1.1, Jumping with Values: Discussion §2.1.3.5.2

2.1.3.5.7 labelled

starter: EXP X,
branches: $\Pi_{i=1}^{n}$ (sh_i: SHAPE,
$branch\_label_i$: LABEL, {n > 0}
tk_i: (TAG sh_i)_OPTION,
$branch_i$: EXP B_i
)
-> EXP (X LUB Y $LUB_{i=1}^{n}$ B_i)

starter is evaluated. If its evaluation runs to completion producing a value, *st*, then *st* is delivered as the result of the whole construct. However, if a *goto(branch_label*$_m$*,exp)* is encountered during the evaluation of *starter*, then the evaluation of *starter* stops. *branch*$_m$ is then evaluated. The result of *exp*, *e*, from the *goto* is bound to tk_m (if supplied) during this new evaluation. This means that inside *body* an evaluation of *obtain_tag(*tk_m*)* will produce the value *e* with SHAPE sh_m. If the evaluation of *branch*$_m$ runs to completion, then the value which it produces, b_m, is delivered as the result of the whole construct.

However, if a *goto(branch_label*$_n$*,exp)* is encountered during the evaluation of *branch*$_m$, then the evaluation of *branch*$_m$ stops. *branch*$_n$ is then evaluated. (*n* may equal *m*.) As before, the value produced by *exp* is bound with tk_n (if supplied) during the evaluation of *branch*$_n$.

Such jumping may continue indefinitely, but if any of the branches' evaluations runs to completion producing a value, *v*, then that value is delivered as the result of the whole construct.

Depending on their run-time behaviour, the result of the construct may be provided by *starter* or one of the *branches*. The SHAPE of the result, which is determined at compile-time, is therefore the LUB of the SHAPES of *starter* and all the *branches*.

The TAGs used for tk_i will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

Translators should suppress the translation of any *branch*$_u$, which can never be evaluated by virtue of the fact that no chain of *gotos* links it with *starter*.

Cross-reference: LABEL §2.1.1.10, Availability of Labels: Discussion §2.1.3.5.1, goto §2.1.3.5.8, Binding: Discussion §2.1.3.1.1, Jumping with Values: Discussion §2.1.3.5.2

2.1.3.5.8 goto

dest: LABEL,
with:EXP X

-> EXP BOTTOM

with is evaluated to produce a value *w*. Control then passes to the LABEL *dest* with the value *w*. This operation will only be used where the LABEL *dest* is available. *dest* will expect a value of SHAPE *X*.

Since the construct can never terminate normally, the SHAPE of its result is *bottom*.

Cross-reference: LABEL §2.1.1.10, Availability of Labels: Discussion §2.1.3.5.1, Jumping with Values: Discussion §2.1.3.5.2

2.1.3.5.9 jump

lab_val: EXP LABEL_VALUE,
with:EXP X

-> EXP BOTTOM

with is evaluated to produce a value *w*. Control then passes to the LABEL represented by *lab_val* with the value *w*. This operation will always be evaluated within the lifetime of the LABEL_VALUE. *lab_val* will have been made from a LABEL which expects a value of SHAPE *X*.

Since the construct can never terminate normally, the SHAPE of its result is *bottom*.

Cross-reference: LABEL §2.1.1.10, Availability of Labels: Discussion §2.1.3.5.1, Lifetimes: Discussion §2.1.3.12, make_label_value §2.1.3.5.11

2.1.3.5.10 return

with:EXP X

-> EXP BOTTOM

with is evaluated to produce a value *w*. The evaluation of the immediately enclosing procedure ceases and the value *w* is delivered as the procedure's result.

Since the construct can never terminate normally, the SHAPE of its result is *bottom*.

An example of the application of 'return' is its use to model the return construct of ANSI C.

Cross-reference: TRACED_PROC §2.3.2.2, UNTRACED_PROC §2.1.2.1.4, procedures §2.1.3.6

2.1.3.5.11 make_label_value

lab: LABEL

-> EXP LABEL_VALUE

A value *lv* is created and delivered which represents the LABEL *lab*. *lv*'s lifetime extends over the construct which introduces *lab*.

In conjunction with *jump*, this construct serves to implement the long jump of ANSI C.

Cross-reference: Availability of Labels: Discussion §2.1.3.5.1, Lifetimes: Discussion §2.1.3.12, LABEL §2.1.1.10, jump §2.1.3.5.9

2.1.3.6 Procedures

2.1.3.6.1 Procedures: Discussion

The treatment of procedures varies considerably from language to language. All have one thing in common - a procedure call is a means of applying the same piece of program to different pieces of data. The TDF *make_untraced_procedure* and *make_traced_procedure* constructs (§2.1.3.6.2, §2.3.3.14) allow one to specify a TAG as the formal parameter and to state its SHAPE; the scope of that TAG is the body of the procedure. If the source language procedure had several parameters, the formal TAG would identify an ALIGNED_TUPLE of the parameters. (See §2.1.2.3.5 for an account of ALIGNED_TUPLE.)

Where languages differ is in treating procedures as data-objects in their own right. Ada, for example, does not allow procedures to be data-objects at all; the only thing that one can do with an Ada procedure is to call it. Pascal allows one procedure to be a parameter of another, but does not allow assignment of procedure values or the delivery of a procedure as the result of another. C allows both of these but restricts the declaration of procedures to a global level. In languages like ML or Lisp, the use of procedures as first-class data objects is of the essence and provides a very effective means of data encapsulation.

These differences are reflected in the way in which the non-local access from within a procedure is compiled. Pascal and Ada were both designed to be implemented on a stack with the accessible non-local stack-frames being transmitted to the procedure at its call in some kind of a display. To access a Pascal non-local, one simply digs it out from one of the stack frames in the display. The target representation of a Pascal procedure as a data-object is simply the address of its code together with the means to construct its display - usually just the stack-frame in which the procedure was declared. In C the situation is simpler since any non-local of a procedure is global.

In these three languages, the TDF representation of a non-local value is likely simply to be a TAG used within the procedure body, declared outside it; the translator is free to translate non-local access by display manipulation. In a language like ML which allows procedures as first-class data-objects, this straightforward approach is likely to be inadequate - it might preclude the re-use of stack-frames, for example. Hence, an ML procedure is likely to be compiled as a closure, represented in TDF as a TUPLE of a POINTER to a set of non-local values and a TDF procedure with no non-local TAGs. A ML procedure call would be translated into TDF as a call of this procedure with the POINTER to the non-locals as an extra parameter. Other intermediate positions are possible; these depend on other choices taken by the compiler, bearing in mind the likely non-local usage in the language.

The use of a non-local TAG in a procedure represents a promise by the compiler that the procedure will not be used outside the evaluation of the body of the declaration

which introduced the TAG (see §2.1.3.12). This applies equally to non-local LABELs - if the procedure body uses a LABEL then the procedure will not be called outside the evaluation of the EXP which introduced the LABEL.

In Pascal, for example, this promise is always valid by reason of the structure of the language. In C, it can be made for non-local values but not for non-local labels introduced by long jump. For other languages, like Algol68, such a promise is not in general valid but may be sustainable in particular cases by analysis of the procedure involved. If a compiler cannot make the promise for non-local values of a procedure, then it can and should produce closures according to the ML model described above.

The situation is slightly different for non-local labels and long jumps. To implement long jumps out of a procedure which might violate the lifetime rules, the compiler should create LABEL_VALUEs (using *make_label_value* §2.1.3.5.11) to be used as a parameter to the *jump* operation (§2.1.3.5.9) in the code of the procedure. (A LABEL_VALUE represents a LABEL as a run-time value.) The LABEL_VALUE could be passed into a procedure as a (global) TAG in C, while in other languages it might be part of the non-locals of a closure. A translator writer concerned to detect whether a jump is to a LABEL which no longer exists can insert a dynamic test to see whether it does (see §2.1.3.5.1).

2.1.3.6.2 make_untraced_procedure

```
param_shape: SHAPE,          {param_shape will be SOME-free}
param: TAG UNTRACED_POINTER(param_shape),
body: EXP X

  -> EXP UNTRACED_PROC
```

Evaluation of *make_untraced_procedure* delivers an UNTRACED_PROC. When this procedure is applied to a parameter using *apply_proc*, space is allocated to hold a value of SHAPE *param_shape*. The value produced by the parameter, which will be of the correct SHAPE, is used to initialise it. *body* is evaluated. During the evaluation, *param* is bound to an original UNTRACED_POINTER pointing to the space. This means that evaluation of *obtain_tag(param)* will produce that POINTER. The value produced by *body* is delivered as the result of the *apply_proc* construct.

The TAG used for *param* will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

TAGs other than *param* which are used in *body* but not declared within it are called non-local TAGs. If and when the procedure is applied and its *body* evaluated, these TAGs obey the same bindings that obtained when the procedure was constructed.

The lifetime of the procedure value is the intersection of the evaluations of the bodies of all the declarations of its non-local TAGs.

If a programming language permits more than one parameter, the compiler to TDF will use *make_untraced_procedure* to construct a TDF procedure whose *param_shape* is ALIGNED_TUPLE (..) (see §2.1.2.3.5). The elements will usually be identified (§2.1.3.1.3).

Cross-reference: UNTRACED_PROC §2.1.2.1.4, Lifetimes: Discussion §2.1.3.12, Binding: Discussion §2.1.3.1.1, Exceptions: Discussion §2.2.3.2.3, apply_proc §2.1.3.6.6, ALIGNED_TUPLE SHAPEs §2.1.2.3.5, identify §2.1.3.1.3

2.1.3.6.3 make_null_untraced_procedure

-> EXP UNTRACED_PROC

A null UNTRACED_PROC is created and delivered. If this PROC is applied, the effect is undefined. The null UNTRACED_PROC may be tested for using *proc_is_null* or *proc_not_null*.

Cross-reference: UNTRACED_PROC §2.1.2.1.4, apply_proc §2.1.3.6.6, proc_is_null §2.1.3.6.4, proc_not_null §2.1.3.6.5

2.1.3.6.4 proc_is_null

not_null:LABEL,
procedure: EXP PROC

-> EXP TOP

procedure is evaluated to produce a TRACED or UNTRACED_PROC value, *p*. If *p* is found to be a null PROC the construct delivers a value of SHAPE *top*. If it is not a null PROC, control passes to the LABEL *not_null* with *top*.

2.1.3.6.5 proc_not_null

is_null:LABEL,
procedure: EXP PROC

-> EXP TOP

procedure is evaluated to produce a TRACED or UNTRACED_PROC value, *p*. If *p* is found not to be a null PROC, the construct delivers a value of SHAPE *top*. If is not a null PROC, control passes to the LABEL *is_null* with *top*.

2.1.3.6.6 apply_proc

result_shape: SHAPE, {result_shape will be SOME-free}
proc: EXP PROC,
arg: EXP X

-> EXP result_shape

proc and *arg* are evaluated to produce values *p* and *a*. The value *p* will either be an UNTRACED_PROC or a TRACED_PROC. It is applied to *a*. The result of this application is delivered as the result of the *apply_proc* construct. It will have SHAPE *result_shape*.

Cross-reference: TRACED_PROC §2.3.2.2, UNTRACED_PROC §2.1.2.1.4

2.1.3.6.7 obtain_current_proc

-> EXP PROC

The procedure currently executing is delivered. It may be TRACED or UNTRACED. When the result of *obtain_current_proc* is supplied as the procedure argument to *apply_proc*, translators should perform a tail-recursion optimisation if this is legitimate.

obtain_current_proc will not be used in contexts where no procedure is running - eg. the outermost level of a TDF capsule.

Cross-reference: TRACED_PROC §2.3.2.2, UNTRACED_PROC §2.1.2.1.4, apply_proc §2.1.3.6.6, Structure of a TDF Capsule §1.6

2.1.3.7 SIZEs and OFFSETs

2.1.3.7.1 shape_size

sh: SHAPE {sh will be SOME-free}

-> EXP SIZE(sh)

A SIZE value is created and delivered which is the size of the SHAPE *sh*. The SHAPE *sh* will be neither bottom nor top and will not contain SOME, unless hidden behind a POINTER. The value delivered by *shape_size* is known at translate time.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.7.2 tuple_size

v: VARIETY,
parts: $\Pi_{i=1}^{n}$ (part$_i$: EXP SIZE(SH$_i$)) {n ≥ 1}

-> EXP TUPLE (TUPLE($\Pi_{i=1}^{n-1}$ OFFSET(TUPLE($\Pi_{j=1}^{i}$(SH$_j$)),SH$_{i+1}$)),
SIZE(TUPLE $\Pi_{i=1}^{n}$ SH$_i$)
)

{SH$_i$ need not be SOME-free}

All *parts* are evaluated to produce SIZE values. A TUPLE *f1* with *n-1* fields is created. Its fields are the OFFSETs of each of the last *n-1* fields of a TUPLE which has components of the sizes given by *parts* from the beginning of the TUPLE (in the given order). A SIZE value *f2* is created which gives the SIZE of a TUPLE which has *n* components of the sizes given by *parts* (in the given order). Finally a TUPLE of *f1* and *f2* is created and delivered.

The OFFSET of the first field of a TUPLE is by definition nought, and so there is no need for *tuple_size* to compute it.

If *n>2* and the same operation is performed for a TUPLE consisting of the first *n-1* fields, the *n-2* OFFSETs resulting shall be the same as the first *n-2* OFFSETs of the original calculation. This implies that adding an extra field at the end shall not affect the positions of the earlier fields.

Given a POINTER to a TUPLE, *pt*, a POINTER to its *i*-th field can be obtained by adding the OFFSET of the *i*-th field, got from *tuple_size*, to *pt* using *add_to_ptr*.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1, add_to_ptr §2.1.3.4.2

2.1.3.7.3 array_size

size: EXP SIZE(X), {X need not be SOME-free}
number: EXP INTEGER(V)

-> EXP SIZE(SOME(X))

size and *number* are evaluated to produce values *s* and *n*. A SIZE value is created and delivered which is the size of the space occupied by an array consisting of *n* copies of a value of size *s*.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.7.4 size_test

ntest: NTEST,
bad: LABEL,
arg1: EXP SIZE(X), {X need not be SOME-free}
arg2: EXP SIZE(Y) {Y need not be SOME-free}

-> EXP TOP

arg1 and *arg2* are evaluated to produce size values, *a* and *b*. These values are compared using the test *ntest*. If the test succeeds, the construct delivers a value of SHAPE *top*. If it fails, control passes to the LABEL *bad* with a value of SHAPE *top*.

Cross-reference: LABEL §2.1.1.10, NTEST §2.1.1.11, TOP §2.1.2.1.2, SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.7.5 bit_size_test

ntest: NTEST,
arg1: EXP SIZE(X), {X need not be SOME-free}
arg2: EXP SIZE(Y) {Y need not be SOME-free}

-> EXP BIT

arg1 and *arg2* are evaluated to produce size values, *a* and *b*. These values are compared using the test *ntest*. If the test succeeds, a *true* BIT is delivered. Otherwise, a *false* BIT is delivered.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.7.6 array_element_offset

sh: SHAPE {sh need not be SOME-free}

-> EXP OFFSET(*sh*,*sh*)

The OFFSET between two adjacent elements in an array (ie. an NOF or a SOME) of values of SHAPE *sh* is calculated and delivered.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1, assign_bits §2.1.3.4.10, contents_bits §2.1.3.4.11

2.1.3.7.7 tuple_element_offset

tuple_sh: SHAPE {sh need not be SOME-free}
sh: SHAPE {sh need not be SOME-free}

-> EXP OFFSET(*tuple_sh,sh*)

The OFFSET of the second element in a value of SHAPE TUPLE(*tuple_sh,sh*) is calculated and delivered.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.7.8 size_in_bytes

v:VARIETY,
arg: EXP SIZE(X) {X need not be SOME-free}

-> EXP INTEGER(v)

arg is evaluated to produce a value, *a*, of SHAPE SIZE(X). The size, *a*, measured in bytes, is delivered as an integer of VARIETY *v*. This target-dependent construct is explicitly required by ANSI C.

Cross-reference: SHAPEs SIZEs and OFFSETs

2.1.3.8 NOFs and SOMEs

2.1.3.8.1 make_nof

parts: $\Pi_{i=1}^{n}$ EXP P {n > 0}

-> EXP NOF(P, N)

The *parts* are evaluated. An NOF is created and delivered which is composed from the values produced, in the same order as they occur in *parts*.

Cross-reference: NOF §1.5.1.3.1

2.1.3.8.2 trim_nof

first: NAT,
number: NAT,
arg:EXP NOF(S, N)

-> EXP NOF(S, N)

arg is evaluated to produce an NOF value, *a*. A new NOF value consisting of *number* components from *a*, starting at *first* is created and delivered as the result of *trim_nof*. Both *first* and *first+number-1* will lie between 1 and *N*.

Cross-reference: NOF §1.5.1.3.1

2.1.3.8.3 concat_nof

arg1:EXP NOF(S, M),
arg2:EXP NOF(S, N)

-> EXP NOF(S, M+N)

arg1 and *arg2* are evaluated to produce values *a* and *b* which are NOFs derived from the same SHAPE, *S*. A new value is created and delivered with SHAPE NOF(S,M+N). Its first M components are copies of the components of *a* and the last N components are copies of the components of *b*.

Cross-reference: NOF §1.5.1.3.1

2.1.3.8.4 and

arg1: EXP S,
arg2: EXP S

-> EXP S {S = NOF(BIT,N) | INTEGER(V) | BIT}

arg1 and *arg2* have the same SHAPE and that SHAPE is the SHAPE of the result. They may be NOF(BIT,N), INTEGER(V) or BIT. They are evaluated to produce values *a* and *b*. The bit-wise intersection of *a* and *b* is delivered as the result.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, Integer SHAPEs §2.1.2.3.1

2.1.3.8.5 or

arg1: EXP S,
arg2: EXP S

-> EXP S {S = NOF(BIT,N) | INTEGER(V) | BIT}

arg1 and *arg2* have the same SHAPE and that SHAPE is the SHAPE of the result. They may be NOF(BIT,N), INTEGER(V) or BIT. They are evaluated to produce values *a* and *b*. The bit-wise union of *a* and *b* is delivered as the result.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, Integer SHAPEs §2.1.2.3.1

2.1.3.8.6 xor

arg1: EXP S,
arg2: EXP S

-> EXP S {S = NOF(BIT,N) | INTEGER(V) | BIT}

arg1 and *arg2* have the same SHAPE and that SHAPE is the SHAPE of the result. They may be NOF(BIT,N), INTEGER(V) or BIT. They are evaluated to produce values *a* and *b*. The bit-wise exclusive or of *a* and *b* is delivered as the result.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, Integer SHAPEs §2.1.2.3.1

2.1.3.8.7 not

arg: EXP S,

-> EXP S {S = NOF(BIT,N) | INTEGER(V) | BIT}

arg has the same SHAPE as the result. It may be NOF(BIT,N), INTEGER(V) or BIT. It is evaluated to produce a value *a*. The bit-wise negation of *a* is delivered as the result.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, Integer SHAPEs §2.1.2.3.1

2.1.3.8.8 n_copies

exp: EXP X,
number: EXP INTEGER(X)

-> EXP SOME(X)

exp and *number* are evaluated to produce values *e* and *n*. A SOME value is created and delivered which contains *n* copies of the value *e*.

Cross-reference: SOME §1.5.1.3.2

2.1.3.9 TUPLEs, ALIGNED_TUPLEs and UNIONs

2.1.3.9.1 make_tuple

parts: $\Pi_{i=1}^{n}$ EXP P_i

-> EXP TUPLE ($\Pi_{i=1}^{n} P_i$)

The *parts* are evaluated. A TUPLE is created and delivered which is composed from the values produced, in the same order as they occur in *parts*.

Cross-reference: TUPLE SHAPEs §2.1.2.3.4, SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.9.2 make_aligned_tuple

parts: $\Pi_{i=1}^{n}$ EXP P_i

-> EXP ALIGNED_TUPLE ($\Pi_{i=1}^{n} P_i$)

The *parts* are evaluated. An ALIGNED_TUPLE is created and delivered which is composed from the values produced, in the same order as they occur in *parts*.

Cross-reference: ALIGNED_TUPLE SHAPEs §2.1.2.3.5, SHAPEs SIZEs and OFFSETs §1.5.1

2.1.3.9.3 field

component: NAT,
tuple: EXP TUPLE ($\Pi_{i=1}^{n} P_i$) {n ≥ 1} {1 ≤ component ≤ n}

-> EXP $P_{component}$

tuple is evaluated to produce a TUPLE value, *t*. The *component*-th field of *t* is delivered as the result of the *field* construct. The SHAPE of the result is the SHAPE of the *component*-th element of *tuple*.

(This construct may also take an ALIGNED_TUPLE argument.)

Cross-reference: TUPLE SHAPEs §2.1.2.3.4, ALIGNED_TUPLE SHAPEs §2.1.2.3.5

2.1.3.9.4 pad

union_shape: UNION $\Pi_{i=1}^{n} X_i$, $\{n > 0\}$
arg: EXP Y $\{\exists k.\ X_k = Y\}$

-> EXP UNION ($\Pi_{i=1}^{n} X_i$)

arg is evaluated to produce a value, *a*. A value of SHAPE *union_shape* is created from *a* and delivered. (This may require the addition of padding.) *arg*'s SHAPE, *Y*, will be one of the X_i.

Cross-reference: UNION §1.5.1.2

2.1.3.9.5 unpad

alt: SHAPE, {alt will be SOME-free}
union: EXP UNION ($\Pi_{i=1}^{n} P_i$) $\{n > 1\}$ $\{\exists k.\ P_k = alt\}$

-> EXP alt

union is evaluated to produce a value *u*. The SHAPE of *u* will be UNION(..) and one of its components will be *alt*. The value of *u* is then delivered, but now with SHAPE *alt*. If *u* in fact has some other SHAPE, the effect is undefined.

Most translators will not generate any code for this operation. It changes the SHAPE of the expression.

Cross-reference: UNION §1.5.1.2

2.1.3.10 Miscellaneous

2.1.3.10.1 make_top

-> EXP TOP

A value of SHAPE *top* is created. This value can be represented by no bits and so no action need be taken to create it.

A value of SHAPE *top* is needed for formal purposes in cases where an EXP must be supplied, but where it is not desired to give any value. An ANSI C procedure taking a void argument, and an ANSI C union having a void field are examples of this need.

Cross-reference: TOP §2.1.2.1.2, apply_proc §2.1.3.6.6, UNION §1.5.1.2

2.1.3.10.2 test_eq

unequal: LABEL
arg1: EXP X,
arg2: EXP X,

-> EXP TOP

arg1 and *arg2* are evaluated to produce values of the same SHAPE, X. The representations of these values are compared. If they are found to be equal the construct delivers a value of SHAPE *top*. If they are found to be unequal, control passes to the LABEL *unequal* with a value of SHAPE *top*.

Translators should, if possible, optimise this construct in cases where either argument is constant. In particular, comparison with the constants *true* or *false* may be common and tokenising these is likely to prove useful.

Cross-reference: LABEL §2.1.1.10, TOP §2.1.2.1.2, tokenisation §1.5.3

2.1.3.10.3 bit_test_eq

arg1: EXP X,
arg2: EXP X,

-> EXP BIT

arg1 and *arg2* are evaluated to produce values of the same SHAPE, X. The representations of these values are compared. If they are found to be equal, a *true* BIT is delivered. Otherwise, a *false* BIT is delivered.

2.1.3.10.4 test_neq

equal: LABEL
arg1: EXP X,
arg2: EXP X,

-> EXP TOP

arg1 and *arg2* are evaluated to produce values of the same SHAPE, X. The representations of these values are compared. If they are found to be unequal the

construct delivers a value of SHAPE *top*. If they are found to be equal control passes to the LABEL *equal* with a value of SHAPE *top*.

Translators should, if possible, optimise this construct in cases where either argument is constant. In particular, comparison with the constants *true* or *false* may be common and tokenising these is likely to prove useful.

Cross-reference: LABEL §2.1.1.10, TOP §2.1.2.1.2, tokenisation §1.5.3

2.1.3.10.5 bit_test_neq

arg1: EXP X,
arg2: EXP X,

-> EXP BIT

arg1 and *arg2* are evaluated to produce values of the same SHAPE, X. The representations of these values are compared. If they are found to be unequal, a *true* BIT is delivered. Otherwise, a *false* BIT is delivered.

2.1.3.10.6 clear_shape

sh:SHAPE

-> EXP *sh*

An empty EXP of SHAPE *sh* is created and delivered.

2.1.3.10.7 exp_evaluated

const: EXP X

-> EXP X

const will be an EXP formed from any combination of the operations listed below, and no other operations. This ensures that it can be completely evaluated at translate-time, leaving no further evaluation necessary at run-time.

make_int §2.1.3.2.2, maxint §2.1.3.2.16, minint §2.1.3.2.17, make_floating §2.1.3.3.1, true §2.2.3.2.10, false §2.2.3.2.11, shape_size §2.1.3.7.1, make_tuple §2.1.3.9.1, make_nof §2.1.3.8.1, pad §2.1.3.9.4, empty_diagnostics §2.2.3.2.6, make_unique_val §2.3.3.8, make_null_whole_pointer §2.3.3.3, make_null_part_pointer §2.3.3.4, integer_to_bits §2.1.3.2.27, whole_to_part §2.3.3.2, pack §2.3.3.11

also the following with SOME-free SHAPE parameters:

size_in_bytes §2.1.3.7.8, size_in_bits §2.2.3.2.9, shape_size §2.1.3.7.1, array_element_offset §2.1.3.7.6, array_size §2.1.3.7.3, union_size §2.2.3.2.16

also make_untraced_procedure(§2.1.3.6.2) and make_traced_procedure(§2.3.3.14) with a procedure of unbounded lifetime (§2.1.3.12).

exp_evaluated is used to indicate that a constant expression which a program requires to be evaluated repeatedly because, say, it lies in a loop, can safely be optimised to being evaluated only once at the outset.

2.1.3.11 Signals: Discussion

ANSI C requires error handling (eg. of overflow) to be done by signals, rather than by exceptions. TDF assumes that a signal handling procedure is supplied for each POSIX process and that procedures can be assigned into places defined by this procedure to handle the signals. The procedure to do this assignment will be identified in TDF by means of an agreed TAG. (See §2.1.1.12.3 for an account of how the relevant signal is raised by an operation which fails.)

In TDF the signals are identified by unique values, whereas in C they are said to be identified by integers (ANSI C 4.7). In a POSIX implementation the uniques must be translated into the appropriate integers.

Otherwise, the rules obeyed by the signal procedures are those specified by ANSI C.

Since clock and asynchronous signals are handled by POSIX, the routines for setting the signal procedures to handle them are also supplied by POSIX.

Cross-reference: Exceptions: Discussion §2.2.3.2.3, Standard Signal §2.1.1.12.3

2.1.3.12 Lifetimes: Discussion

This is a convenient point at which to introduce the concept of lifetime and discuss its importance to writers of TDF translators.

A danger in C and other languages is the use of a pointer to a space which is no longer alive, in the sense that the space pointed to is on a stack and has been re-used for some other purpose. Such mistakes can be very hard to find. Like C, TDF permits this mistake to be made and says that the effect is undefined.

TDF defines rules about the lifetimes of values and LABELs, which may be assumed to hold by writers of translators from TDF to machine code. These rules permit (but do not require) a conventional stack implementation of variables in TDF. TDF derived from a language such as ML will obey these rules, since they are enforced by an ML compiler. TDF produced from other sources such as a C compiler

may not obey them, usually because the program being compiled is wrong but the compiler was not able to detect the mistake.

Lifetime is a property of a value or LABEL. A lifetime is defined to be either unbounded or to extend over the time during which a certain EXP is being evaluated. Thus lifetimes form a partial ordering by simple nesting.

- The lifetime of any scalar value is unbounded.
- The lifetime of any POINTER whose original POINTER (§2.1.3.4.1.3) was created by *generate* is unbounded.
- The lifetime of any POINTER created by a library routine is undefined.
- The lifetime of any POINTER whose original POINTER was created by *variable* or *variable_no_init* extends only over those constructs.
- The lifetime of a compound value is the intersection of the lifetimes of its component values. This means that it is equal to the lifetime of the shortest-living component.
- The lifetime of a procedure value is the intersection of the bodies of the declarations of its non-local TAGs and the EXPs in which the non-local LABELs which it uses are available.
- The lifetime of a LABEL extends over the evaluation of the *labelled*, *conditional* or *repeat* which introduces the LABEL.

The basic condition for a stack implementation to be correct is that no access be made to a value outside of its lifetime.

This may be assumed to hold by TDF translator writers. It is a dynamic condition which is impracticable (and usually undesirable) to check at run-time. Static rules can be formulated which are sufficient to ensure that the only values accessible at any one time have lifetimes which form a total ordering, allowing a stack implementation of nested declarations. These rules are:

- In any assignment, the lifetime of the right-hand side is greater than or equal to the left-hand side.
- In any procedure, the lifetime of its result value is greater than or equal to the lifetime of the procedure.

In total generality, however, the application of these rules is difficult (if not impossible). Languages like Ada and Pascal avoid the problem by disallowing both the assignment (and delivery as results) of procedures or local pointers. In C, leaving aside long-jumps, the lifetime of all procedures is unbounded (their only non-locals

are global) and so there is no difficulty about assigning or delivering procedures. However C's & operator allows one to assign and deliver local pointers. This means that although blatant errors can be detected relatively easily at translate time, mis-use via parameters of procedures cannot. Usually the first intimation of the error is a collapse of the run-time system when some crucial stack-pointer is overwritten because the space to which it points is being re-used.

In some languages (eg ML, Lisp) every value must have an unbounded lifetime. This has two consequences for the TDF to which they compile:

1. The only possible use of a TAG declared by a variable declaration is in *contents*, as the rvalue of an *assign*, or some related limited use. If another use were required, the TAG would have to be bound to the result of a *generate* (possibly operated on by *whole_to_part*) using an *identify* declaration. This is equivalent to a variable declaration except that space is taken from heap space rather than from a stack frame; the value bound to the TAG has an unbounded lifetime.

2. Only global TAGs are used as non-locals of procedures; often no TAGs would be used as non-locals. An ML procedure, for instance, would probably be represented as a closure formed from a global POINTER, its non-local values and a TDF procedure (with no non-local TAGs) which needs this POINTER as an extra parameter over and above the parameters specified in the ML. This treats the ML procedure rather like a partial application of a global procedure which has the non-locals as parameters.

1. and 2. together mean that the non-locals have an unbounded lifetime.

This method of "unbounding" values does not extend to LABEL_VALUEs - they can never have an unbounded lifetime. This is of little consequence to languages which need not use them (like ML or Ada) or which do need to use them but never in a manner which could violate the lifetime rules (like COBOL or Pascal). Some languages (like Algol68 or C) do allow "long jumps" out of procedures to "dead" LABELs (ones whose stack_frames have been exited from). If one wished to protect against this, a run-time check would be necessary (see §2.1.3.5.1).

In a garbage collecting TDF implementation, a POINTER produced by *generate* continues to be usable as long as it is accessible. A mistake similar to that described above can be made by de-allocating and subsequently using such a POINTER. However, the TDF concept of lifetime does not have any bearing on this mistake.

2.1.4 tokenise and TOKEN Application

As explained in §1.6, *tokenise* is a program construct which serves to identify (possibly parameterised) pieces of TDF program. It is analogous to *identify* (see §2.1.3.1.3), but whereas *identify* identifies values at run-time, *tokenise* identifies pieces of program at translate-time.

It has the form:

k: SORT,
token: TOKEN,
pars: $\Pi_{i=1}^{n}$ (t_i:TOKEN, k_i:SORT),
def: k,
body: EXP X

-> EXP X

When the *tokenise* construct is translated, only *body* is translated to machine code. However, within *body* any occurrence of *token* is taken to stand for the program fragment *def*, which will be of SORT *k*. The TOKEN may be parameterised. If it is, the SORTs of the parameters which it expects are described by the k_i in *pars*. Within the program fragment *def*, the parameters are bound with the TOKENs t_i. (The TOKENs t_i will not themselves be parameterised.) The TOKENs t_i will be used only within *def*. The TOKEN *token* will be used only within *body* with one exception: *token* may occur within *def* in order to form the definition of a circular SHAPE. The TOKENs used in all the parameters will be disjoint from the t_i and from all the TOKENs used in *def*. There will be no reuse of TOKENs and no scoping rule for TOKENs is defined.

(TOKEN is not counted as one of the SORTs of TDF, since a TOKEN indicates a substitution to produce a SORT, rather than a SORT itself.)

Cross-reference: Structure of a TDF Capsule §1.6, TDF: Scenario of Use §1.1

In order to obtain the program fragment for which a TOKEN stands, TDF constructs *apply_exp_token*, *apply_shape_token* etc. are provided - one for each different SORT. They all have the same form. Using *apply_exp_token* as an example:

token: TOKEN,
pars: $\Pi_{i=1}^{n}$ (k_i:SORT)

-> EXP X

apply_exp_token takes two arguments: first a TOKEN which will stand for an EXP X; and second, and a number of program fragments, *pars*, which may be of any SORT, but which will conform to *token*'s parameter requirements set out when it was

defined. The EXP for which *token* stands is the result and can then be translated just like any other EXP. This process is analogous to the application of the C preprocessor.

2.1.5 Constructs for Conditional Compilation

As explained in §1.5.2, two TDF constructs allow the conditional translation of program fragments depending on the value delivered by an EXP evaluated at translate-time. These are *exp_cond* and *variety_cond*.

2.1.5.1 exp_cond

control: EXP INTEGER(V),
exp1: EXP X,
exp2: EXP X

-> EXP X

At **translate-time**, *control* is evaluated to produce a value, *c*. If *c* is non-zero, then the program fragment *exp1* is selected for translation. If *c* is zero, then the program fragment *exp2* is selected.

2.1.5.2 variety_cond

control: EXP INTEGER(V),
v1: VARIETY,
v2: VARIETY

-> VARIETY

At **translate-time**, *control* is evaluated to produce a value, *c*. If *c* is non-zero, then the program fragment *v1* is selected for translation. If *c* is zero, then the program fragment *v2* is selected.

2.2 TDF Level 1

This section defines the extra SORTs, SHAPEs and EXPs which together with the Level 0 constructs fonn TDF Level 1 - ie. the subset of full TDF which is required in order to implement ANSI C, Ada and other languages which do not require garbage collection. Level 1 goes beyond Level 0 in areas such as provision for parallel processes and diagnosis of exceptions.

2.2.1 Level 1 SORTs

There are thirteen SORTs in TDF, twelve of which form part of Level 0. The thirteenth - EXCEPTION_HANDLER - appears in Level 1.

2.2.1.1 EXCEPTION_HANDLER

A value of SORT EXCEPTION_HANDLER contains a collection of EXPs which may be evaluated in the event of an exception being encountered. Its form is complex:

(special_handlers: $\Pi_{i=1}^{n}$ (exception_unique$_i$: UNIQUE,
pt$_i$: (TAG UNTRACED_POINTER(W$_i$))_OPTION,
dt$_i$: (TAG DIAG)_OPTION,
body$_i$: EXP X$_i$
),
default_handler: (def_et: (TAG EXCEPTION_VALUE)_OPTION,
def_dt: (TAG DIAG)_OPTION,
def_body: (EXP Y)_OPTION
)
)

Each EXCEPTION_HANDLER has an associated SHAPE, the SHAPE of the value which it will produce if brought into play by the construct *handle_exception*. This SHAPE is the LUB of the SHAPEs of all the values which it might produce:

$$LUB_{i=1}^{n} X_i \text{ LUB } Y$$

The use of EXCEPTION_HANDLERs is explained in the account of *handle_exception* (§2.2.3.2.7).

One ERROR_TREATMENT not available in Level 0 becomes available in Level 1. It is *standard_exception*.

2.2.1.2 Standard Exception

This ERROR_TREATMENT is used when an error can occur and the desired effect is to produce an exception (see §2.2.3.2.3).

2.2.2 Level 1 SHAPEs

Level 1 contains two SHAPEs not found in Level 0. These are:

THREAD
DIAG

They are both described below:

2.2.2.1 THREAD

Implementations of programming languages' process models in terms of TDF's lightweight processes require administrative information to be held as part of their process data. The TDF SHAPE of this information is THREAD.

In a stack-based implementation a THREAD is likely to be just a stack. In a heap-based implementation it is likely to be a chain of workspaces.

2.2.2.2 DIAG

The SHAPE describing the parcel of information which is made available when a program fails. (DIAG stands for 'diagnostics'.)

Users cannot create DIAG values, except for the empty DIAG with which to start a failure. Only the system (ie. code produced by the translator) can create other DIAG values. However, the user can operate on DIAG values in order to determine what went wrong. In order to avoid reverse engineering, diagnostic information is given in terms of UNIQUE values, and the connection between these UNIQUE values and the structure and identifiers of the original program is made separately.

Cross-reference: Exceptions: Discussion §2.2.3.2.3, empty_diagnostics §2.2.3.2.6

As well as introducing two new SHAPEs, THREAD and DIAG, Level 1 specifies that translators should implement VARIETYs and FLOATING_VARIETYs sufficient to accommodate the requirements of Ada.

This is a convenient point at which to give an account of the way in which SIZEs and OFFSETs relate to Ada.

2.2.2.3 SHAPEs SIZEs and OFFSETs: Ada

Space-efficient implementation of Ada requires dynamic (run-time) variability of sizes, even within tuples. An Ada record, for example, may contain fields whose space requirements depend on the discriminants of the record.

Consider the Ada record given by:

```
type rec(d:integer) is
 record  first:string(1..d);
         second:string(1..d);
 end record;
```

TDF's SIZE construct provides the means for handling its potentially varying space requirement. A space-saving mapping of this record to a TDF SHAPE could be:

```
WHOLE_POINTER
    (TUPLE
      (best_signed,    -- the discriminant d
       x,              -- an offset to field second
       SIZE(SOME(byte)), -- size of field first
       SIZE(SOME(byte)), -- size of field second
       SOME(byte),     -- field first
       SOME(byte)      -- field second
      )
    )
```

where *x* and *byte* are both tokenised SHAPEs. *byte* would probably be defined as:

```
byte = INTEGER(0, 255)
```

and x would be recursively defined as:

```
x = OFFSET(TUPLE(best_signed,
                 x,
                 SIZE(SOME(byte)),
                 SIZE(SOME(byte)),
                 SOME(byte)
                ),
           SOME(byte)
          )
```

The value of the second element in the TDF TUPLE is dynamically computed using the construct *tuple_element_offset* (§2.1.3.7.7). To access the string field *second* in the Ada record one would use *add_to_ptr* on the original POINTER to the TUPLE and the OFFSET value to obtain a POINTER to the field *second*. (Accessing the string field *first*

is simpler because its OFFSET from the start of the TUPLE can be calculated at translate-time.)

This implementation means that the space occupied by an instance of the record is a function of the discriminant value.

The TDF TUPLE implementing the Ada record *rec* has to be hidden behind a POINTER whenever it is used in a SOME-free position, e.g. when a value of this SHAPE is declared. The advantage of the TDF representation is that it uses the minimum amount of space whilst still allowing simple operations on the flat representation of the TUPLE to determine properties such as equality between records of this SHAPE.

By contrast, many current commercial Ada compilers cannot handle a record such as *rec* because they adopt a strategy of reserving the maximum amount of space for the fields *first* and *second* that the discriminant might allow. Unfortunately in this case the discriminant can be up to maxint, which will mean that there is insufficient memory in the machine!

Notice that TDF does not provide special SHAPEs for datastructures which require complex implicit descriptors. All complex structures, such as the Ada record just described or multi-dimensional arrays, are implemented using the very general facilities offered by TDF's SHAPEs. In the example of the Ada record *rec*, the first three fields of the TUPLE constitute the descriptor.

2.2.3 Level 1 EXPs

The new constructs added in Level 1 can conveniently be divided into two categories:

Lightweight Processes
Constructs to Support Ada

(Ada is highlighted here because it is the richest of the languages that can compile to TDF Level 1.) The two categories are described in the following sections. (The notation used for describing the constructs is introduced in §1.7.)

2.2.3.1 Lightweight Processes

2.2.3.1.1 Lightweight Processes: Discussion

For the purposes of ANSI C, TDF runs in POSIX processes. However, POSIX

processes are costly in terms of space and speed of process swapping and process creation and are not a good basis for the implementation of Ada processes. TDF is therefore planned to provide for the creation of lightweight processes within a POSIX process.

Many of the facilities used by programming languages have settled down over the years, so that a set of operations of reasonable size covers their requirements fairly closely. But this is not the case for processes and so TDF's process mechanism is defined at a low level, so that the very different process mechanisms of existing languages can be accommodated. This implies that much of the implementation of Ada processes has to be supplied as an explicit implementation in TDF.

POSIX handles clock and asynchronous signals and supplies the routines for setting the signal procedures for dealing with them. These will be identified in TDF by agreed TAGs.

For multiple processors using a common memory, finding an architecture neutral description of processes is awkward because the mechanisms used by the hardware to implement locking are not uniform. The operations described below, *test_and_set* and *test_and_clear*, can be implemented on many machines though they are not necessarily the most efficient operations for every machine. This area is developing at the moment and it has been judged best to retain these operations for the meantime but leave the area open for later re-consideration.

Multiple processors running only POSIX processes do not present any problem since the interactions are very limited and POSIX can be assumed to handle them satisfactorily.

Cross-reference: test_and_clear §2.2.3.1.6, test_and_set §2.2.3.1.5, discard_thread §2.2.3.1.4, exchange_thread §2.2.3.1.3, create_thread §2.2.3.1.2

2.2.3.1.2 create_thread

proc: EXP PROC

-> EXP THREAD

proc is evaluated to produce a procedure *p*, which may be TRACED or UNTRACED. The procedure *p* will expect a THREAD parameter and deliver a TOP result. A new THREAD *t* is created from *p*. If and when *t* is started (by *exchange_thread*) it will behave as if it were evaluating the procedure *p*, with *p's* parameter being supplied by the *exchange_thread* operation. The procedure will be one which, if applied using *apply_proc*, would never complete.

Cross-reference: THREAD §2.2.2.1, exchange_thread §2.2.3.1.3, Lightweight Processes: Discussion §2.2.3.1.1

2.2.3.1.3 exchange_thread

thread: EXP THREAD

-> EXP THREAD

thread is evaluated to produce a THREAD value, *t*. The execution of the current process whose THREAD is *c* is stopped and the execution of the process represented by *t* resumes or begins.

If the reason that *t* was not executing was that it had halted with an *exchange_thread* operation, then the current THREAD *c* is delivered as the result of that *exchange_thread* operation in *t*.

If *t* had just been created by *create_thread*, the THREAD of the newly stopped process is supplied as if it were the parameter of the procedure from which *t* was created.

Cross-reference: THREAD §2.2.2.1, create_thread §2.2.3.1.2, Lightweight Processes: Discussion §2.2.3.1.1

2.2.3.1.4 discard_thread

thread: EXP THREAD

-> EXP TOP

thread is evaluated to produce a THREAD value, *t*. The THREAD *t* is discarded. This operation should require no work on a ga-bage collected system which implements processes with the heap.

Cross-reference: THREAD §2.2.2.1, create_thread §2.2.3.1.2, Lightweight Processes: Discussion §2.2.3.1.1

2.2.3.1.5 test_and_set

lab: LABEL
ptr: EXP POINTER(INTEGER(best_signed))

-> EXP TOP

ptr is evaluated to produce a POINTER, *p*. An integer of SHAPE INTEGER(BEST_SIGNED) will lie in the space pointed to by the *p*. Its value will be either 0 or 1. It is tested. If it is 1, control passes to the LABEL *lab*. If it is 0, control

does not pass elsewhere and a value of SHAPE *top* is delivered. In any case the value 1 is written into the space pointed to by *p*.

This operation is interlocked against operations by other processors accessing the same space.

Cross-reference: THREAD §2.2.2.1, Lightweight Processes: Discussion §2.2.3.1.1

2.2.3.1.6 test_and_clear

lab: LABEL
ptr: EXP POINTER(INTEGER(best_signed))

-> EXP TOP

ptr is evaluated to produce a POINTER, *p*. An integer of SHAPE INTEGER(BEST_SIGNED) will lie in the space pointed to by the *p*. Its value will be either 0 or 1. It is tested. If it is 1, control passes to the LABEL *lab*. If it is 0, control does not pass elsewhere and a value of SHAPE *top* is delivered. In any case the value 0 is written into the space pointed to by *p*.

This operation is interlocked against operations by other processors accessing the same space.

Cross-reference: THREAD §2.2.2.1, Lightweight Processes: Discussion §2.2.3.1.1

2.2.3.2 Constructs to Support Ada

2.2.3.2.1 equal_contents

arg1, *arg2* and *arg*3 are evaluated to

unequal: LABEL,
arg1: EXP POINTER(A),
arg2: EXP POINTER(B),
arg3: EXP SIZE(X) {X need not be SOME-free}

-> EXP TOP

produce values, *a*, b and *c*. *a* and *b* will be POINTERs. *c* will be a SIZE. The amount of data specified by *c* located in the spaces pointed to by *a* and *b* is compared for equality of representation. If they are equal the construct delivers *top*. If they are unequal control passess to the LABEL *unequal* with *top*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, SHAPEs SIZEs and OFFSETs §1.5.1, LABEL §2.1.1.10

2.2.3.2.2 not_equal_contents *arg1*, *arg2* and *arg3* are evaluated

equal: LABEL,
arg1: EXP POINTER(A),
arg2: EXP POINTER(B),
arg3: EXP SIZE(X) {X need not be SOME-free}

-> EXP TOP

to produce values, *a*, b and *c*. *a* and *b* will be POINTERs. *c* will be a size. The amount of data specified by *c* located in the spaces pointed to by *a* and *b* is compared for equality of representation. If they are unequal the construct delivers *top*. If they are equal control passes to the LABEL *equal* with *top*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, SHAPEs SIZEs and OFFSETs §1.5.1, LABEL §2.1.1.10

2.2.3.2.3 Exceptions: Discussion

TDF exceptions are used to implement Ada and ML exceptions, among other constructs. Since TDF can be used to write operating systems, the TDF model of exceptions is able to handle diagnostics as data.

2.2.3.2.3.1 EXCEPTION_VALUEs

EXCEPTION_VALUE is a convenient term for describing the parcel of information arising from the occurrence of an exception. An EXCEPTION_VALUE consists of a pair of values, the first of which is also a pair.

EXCEPTION_VALUE = (EXCEPTION_IDENTIFIER, DIAG)

EXCEPTION_IDENTIFIER = (EXCEPTION_UNIQUE, UNTRACED_POINTER(X))

An EXCEPTION_UNIQUE is a UNIQUE value, used to characterise a class of exceptions. The following six are pre-defined:

2.2.3.2.3.1.1 overflow
2.2.3.2.3.1.2 divide_by_zero
2.2.3.2.3.1.3 store_full
2.2.3.2.3.1.4 null_pointer
2.2.3.2.3.1.5 bound_check
2.2.3.2.3.1.6 absent_shaky

but these may be added to, either statically by agreement between groups of users, or dynamically during the running of a program.

The POINTER serves to give extra information to an EXCEPTION_HANDLER. For example, IEEE floating point defines certain floating point numbers which give more information about a failed operation. Different classes of exception will usually require different kinds of extra information. In order that EXCEPTION_VALUEs can be represented in a uniform way, all these kinds of extra information are reduced to one shape, UNTRACED_POINTER, which will point to space holding the information.

The DIAG value serves to give information about the state of the procedures which were active between the point at which the exception originated and the point at which it is being handled. This information, in combination with information derived from the original program, will allow the values being used within these procedures to be examined according to the conventions of the originating program. Various methods exist of controlling the amount of information which is accessible, in order to discourage reverse engineering, while still permitting programs to be diagnosed (see Diagnostics §2.2.3.2.3.3). See also §2.1.3.1.4 for a discussion of the *visible* qualifier.

An EXCEPTION_VALUE can originate from an ERROR_TREATMENT or from the *fail* or *fail_no_diag* constructs.

If it originates from an ERROR_TREATMENT, the UNIQUE is determined by the operation which caused the error and the POINTER value by the details of the error, in a way specified in the operation. In this case the DIAGNOSTICS starts as empty.

If it originates from *fail* or *fail_no_diag*, the EXCEPTION_IDENTIFIER and the DIAG are explicitly supplied as arguments. In particular, the diagnostics may be *empty_diagnostics* or may be a diagnostic value which already exists. By this means, exceptions may be tested to see if a handler wishes to deal with them.

Cross-reference: diagnostics §2.2.3.2.3.3, EXCEPTION_HANDLER §2.2.1.1, empty_diagnostics §2.2.3.2.6, fail §2.2.3.2.4, fail_no_diag §2.2.3.2.5, ERROR_TREATMENT §2.1.1.12

2.2.3.2.3.2 Propagation of Exceptions

All constructs except for *handle_exception* treat exceptions in the same way. When an exception occurs, a search is made for the closest dynamically enclosing *handle_exception* construct. This may be found in the current procedure, or in the procedure from which the current one was called, or in the procedure which called that one and so on right back to the main procedure in the current process. As each procedure level is exited, information about that procedure is added to the diagnostic chain, modifying the EXCEPTION_VALUE. Once an enclosing *handle_exception* is found which has as argument an EXCEPTION_HANDLER capable of handling the

exception, the EXCEPTION_VALUE's POINTER and DIAG components are bound to TAGs specified by the EXCEPTION_HANDLER and the EXCEPTION_HANDLER takes control.

If it is desired to prevent part of this chain from being visible in diagnostic information, a *handle_exception* construct can be inserted which re-fails using *fail_no_diag*. The consequence will be that any subsequent diagnosis of an exception will not be able to obtain any diagnostic information from the procedure in which the *handle_exception* has been put. This can be useful, for instance, if a procedure which requires concealment calls a user procedure which requires diagnostics. An example might be a differential equation solver.

The propagation of exceptions has been described as if it were performed by an interpreter, but there is no implication that this is how it should be done. Any equivalent scheme - for instance, one which detects what exceptions are being tested and jumps directly to the correct place - can be implemented.

Cross-reference: exception_handler §2.2.1.1, fail_no_diag §2.2.3.2.5

2.2.3.2.3.3 Diagnostics

Diagnostic information, where it is provided, is associated with the activation of a procedure. It consists of two parts, one compulsory, the other optional.

The compulsory part is a sequence of UNIQUEs corresponding to the *diagnose_point* constructs which were encountered on the scan back to the *handle_exception* construct.

The optional part is a set of pairs of TAGs and values corresponding to the *identify*, *variable* and *variable_no_init* constructs which are active in the procedure. At least those TAGs which are declared to be *visible* shall appear in the set, together with as many others as are available.

If only the compulsory diagnostics are available, a sequence of UNIQUE values partially identifying the location of the error can be sent back to the distributor.

Cross-reference: identify §2.1.3.1.3, variable §2.1.3.1.4, handle_exception §2.2.3.2.7

2.2.3.2.4 fail

exception_id: EXP EXCEPTION_IDENTIFIER,
diag: EXP DIAG

-> EXP BOTTOM

exception_id is evaluated to produce an EXCEPTION_IDENTIFIER, *e* and *diag* to

produce a DIAG, *d*. An exception is created using *e* and *d*. Since this construct does not terminate normally, the SHAPE of its result is BOTTOM.

Cross-reference: Exceptions: Discussion §2.2.3.2.3

2.2.3.2.5 fail_no_diag

exception_id: EXP EXCEPTION_IDENTIFIER

-> EXP BOTTOM

exception_id is evaluated to produce an EXCEPTION_IDENTIFIER, *e*. An exception is created using *e* and an empty diagnostic chain. Since this construct does not terminate normally, the SHAPE of its result is BOTTOM.

Cross-reference: Exceptions: Discussion §2.2.3.2.3

2.2.3.2.6 empty_diagnostics

-> EXP DIAG

An empty diagnostic chain is created and delivered.

Cross-reference: Exceptions: Discussion

2.2.3.2.7 handle_exception

body: EXP X,
handler: EXCEPTION_HANDLER Y

-> EXP (X LUB Y)

body is evaluated. If its evaluation completes successfully, its result is delivered as the result of *handle_exception*, with SHAPE (X LUB Y).

If the evaluation of *body* does not complete successfully, but instead produces an EXCEPTION_VALUE, then *handler* comes into play. (The reader may wish to refer to §2.2.1.1 at this point.) The UNIQUE, *u*, from the EXCEPTION_VALUE is tested against each of the $exception\_unique_i$ contained in *handler*. If and when a match is found between *u* and one of the $exception\_unique_i$, then the POINTER, *p*, and the DIAG, *d*, from the EXCEPTION_VALUE are bound with the TAGs pt_i and dt_i (if present). $body_i$ from the EXCEPTION_HANDLER is then evaluated and its result delivered with SHAPE (X LUB Y).

If, however, no match is found for *u*, but the EXCEPTION_HANDLER's optional

default_handler is present, the following actions are performed. The EXCEPTION_VALUE and the DIAG, *d*, are bound with the TAGs *def_et* and *def_dt*. Then *def_body* is evaluated and its result delivered with SHAPE (X LUB Y).

If the EXCEPTION_HANDLER's optional *default_handler* is **not** present, then the EXCEPTION_VALUE is simply passed on to the next enclosing *handle_exception* construct (if any) to see if it can handle the exception.

Cross-reference: Least Upper Bound §2.1.2.2, Exceptions: Discussion §2.2.3.2.3, EXCEPTION_HANDLER §2.2.1.1, EXCEPTION_VALUEs §2.2.3.2.3.1

2.2.3.2.8 select_from_nof

arg: EXP NOF(X, N),
index: EXP INTEGER(V)

-> EXP X

arg is evaluated to produce an NOF(X, N) value, *a*. *index* is evaluated to produce an INTEGER(V) value, *i*, which will lie between 0 and N-1 inclusive. The *i*-th component of *a* is delivered.

Cross-reference: NOF §1.5.1.3.1

2.2.3.2.9 size_in_bits

v: VARIETY,
arg: EXP SIZE(X) {X need not be SOME-free}

-> EXP INTEGER(v)

arg is evaluated to produce a SIZE value, *s*. An integer value of VARIETY *v* is created and delivered which is the number of bits occupied by a value of the size *s*.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1

2.2.3.2.10 true

-> EXP BIT

A *true* BIT value is created and delivered.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.2.3.2.11 false

-> EXP BIT

A *false* BIT value is created and delivered.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.2.3.2.12 test_eq_bit

arg1: EXP X,
arg2: EXP X

-> EXP NOF(BIT, 1)

arg1 and *arg2* are evaluated to produce *a* and *b*, values of the same SHAPE. The representations of *a* and *b* are compared. An NOF(BIT,1) value is created and delivered which contains *true* if they are equal and *false* if they are not.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.2.3.2.13 integer_test_bit

ntest: NTEST,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP NOF(BIT, 1)

arg1 and *arg2* are evaluated to produce *a* and *b*, integers of the same VARIETY. These integers are compared using the test *ntest*. An NOF(BIT,1) value is created and delivered. It contains *true* if the test succeeds and is produced *false* if it does not.

For example, if *ntest* is *greater*, then if *a* is greater than *b* the construct yields *true*. Otherwise it yields *false*.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, NTEST §2.1.1.11

2.2.3.2.14 floating_test_bit

ntest: NTEST,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

-> EXP NOF(BIT, 1)

arg1 and *arg2* are evaluated to produce *a* and *b*, floating point numbers of the same FLOATING_VARIETY. *a* and *b* are compared using the test *ntest*. An NOF(BIT,1) value is created and delivered. It contains *true* if the test succeeds and *false* if it does not.

For example, if *ntest* is *greater*, then if *a* is greater than *b* the construct yields *true*. Otherwise it yields *false*.

Cross-reference: NOF §1.5.1.3.1, BIT §2.1.2.1.3, NTEST §2.1.1.11

2.2.3.2.15 equal_contents_test_bit

arg1: EXP POINTER(Y),
arg2: EXP POINTER(Y),
arg3: EXP SIZE(X) {X need not be SOME-free}

-> EXP NOF(BIT, 1)

arg1, *arg2* and *arg3* are evaluated to produce values, *a*, b and *c*. *a* and *b* will be POINTERs. *c* will be a SIZE. The amount of data specified by *c* located in the spaces pointed to by *a* and *b* is compared for equality of representation. An NOF(BIT,1) value is created and delivered. It contains *true* if the representations were found to be equal and *false* if they were not.

Cross-reference: Pointers: Discussion §2.1.3.4.1, SHAPEs SIZEs and OFFSETs §1.5.1, NOF §1.5.1.3.1, BIT §2.1.2.1.3

2.2.3.2.16 union_size

alts: $\Pi_{i=1}^{n}$ (part$_i$: EXP SIZE(SH$_i$)) {SH$_1$ need not be SOME-free}
-> EXP SIZE(UNION ($\Pi_{i=1}^{n}$ SH$_i$))

alts are evaluated to produce SIZE values. A new SIZE value is created and delivered which is the size that a UNION composed of values of these SIZEs would have.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1

2.2.3.2.17 index

ptr: EXP UNTRACED(X),
offset: EXP OFFSET(X,X),
low: EXP INTEGER(V),
high: EXP INTEGER(V),
index: EXP INTEGER(V)

-> EXP UNTRACED_POINTER(X)

ptr, *offset*, *low*, *high* and *index* are evaluated to produce values *p*, *off*, *lo*, *hi* and *i* respectively. *i* is checked to see whether it lies between *lo* and *hi* inclusive. If it does not, a *bound_check* exception is produced. But if it does, then *off* is scaled by *i* and the value delivered as the result of the construct is the sum of the resulting OFFSET and the POINTER value *p*.

If *p* delivers a null POINTER, the effect is undefined.

The result will share with *p*.

This operation is provided to take advantage of machines which have specific instructions for indexing. It is equivalent to a combination of *add_to_ptr*, *mult* and *integer_test*. Where this combination occurs directly the *index* operation should always be used in preference.

Note that in Level 2 a WHOLE_POINTER(X) argument can be provided, in which case the SHAPE of the result is PART_POINTER(X).

Cross-reference: Pointers: Discussion §2.1.3.4.1, add_to_ptr §2.1.3.4.2, Exceptions: Discussion §2.2.3.2.3, Lifetimes: Discussion §2.1.3.12, null POINTERs §2.1.3.4.1.2, mult §2.1.3.2.5, integer_test §2.1.3.2.25

2.2.4 Level 1 Constructs for Conditional Compilation

For the purposes of ANSI C, the only conditional compilation constructs required are *exp_cond* and *variety_cond*. Level 1 includes similar constructs covering all the other SORTs:

2.2.5 shape_cond
2.2.6 nat_cond
2.2.7 signed_nat_cond
2.2.8 floating_variety_cond
2.2.9 bool_cond
2.2.10 unique_cond

2.2.11 tag_cond
2.2.12 label_cond
2.2.13 ntest_cond
2.2.14 error_treatment_cond
2.2.15 exception_handler_cond

They all have the form:

control: EXP INTEGER(V),
s1: SORT,
s2: SORT

-> SORT

and behave in a similar manner to *exp_cond* and *variety_cond*.

Cross-reference: Conditional Compilation §1.5.2, Constructs for Conditional Compil. tion §2.1.5

2.3 TDF Level 2

This section defines the SORTs, SHAPEs and EXPs which go together to form TDF Level 2 - ie. full TDF. Level 2 goes beyond Level 1 in offering POINTER operations which interact with a garbage collector, means for creating UNIQUE values, support for unanticipated procedure application and a number of other miscellaneous features.

2.3.1 Level 2 SORTs

All thirteen TDF SORTs are included in TDF Levels 0 and 1. Level 2 therefore contains no more SORTs than the thirteen available introduced in Level 1:

EXP	UNIQUE
SHAPE	TAG
NAT	LABEL
SIGNED_NAT	NTEST
VARIETY	ERROR_TREATMENT
FLOATING_VARIETY	EXCEPTION_HANDLER
BOOL	

2.3.2 Level 2 SHAPEs

All the SHAPEs of the previous sections shall be implemented, plus the following.

2.3.2.1 POINTER SHAPEs Concerned with Garbage Collection

There are five SHAPE constructs collectively known as POINTERs. They are:

UNTRACED_POINTER
WHOLE_POINTER
SHAKY_WHOLE_POINTER
PART_POINTER
SHAKY_PART_POINTER

UNTRACED_POINTER forms part of Level 0 and was introduced in §2.1.2.3.3. All the others are concerned with garbage collection and hence form part of Level 2.

2.3.2.1.1 Garbage Collection: Discussion

As mentioned in §1.1 there is a difference in the memory management for TDF programs between levels 0 and 1, and level 2. Level 2 differs by supporting automatic garbage collection.

Although it encompasses a garbage collected memory, Level 2 also supports the Level 0 notion of an untraced kernel which a garbage collector ignores. POINTERs and PROCs in the untraced kernel are of SHAPEs UNTRACED_POINTER and those constructed from UNTRACED_PROC. (These SHAPEs are referred to as untraced because the garbage collector - if present - does not trace an UNTRACED_PROC's workspace or the contents of an UNTRACED_POINTER.)

Levels 0 and 1 are totally implemented in the untraced kernel. However, in Level 2 TDF issues of communication between the traced world and the untraced kernel arise. No value of a traced SHAPE can be accessed in the untraced kernel. In other words, there can be no POINTERs pointing from the traced world to the untraced kernel.

2.3.2.1.2 WHOLE_POINTER SHAPEs

A WHOLE_POINTER is a pointer which points to the whole of an allocated space. The value delivered by *generate* has SHAPE WHOLE_POINTER(..).

WHOLE_POINTERs have equal representation if and only if they are identical - ie. they are copies of a value produced from one particular evaluation of *generate*.

In garbage collected systems it may be desirable to represent WHOLE and PART_POINTERS differently. Typically in a garbage collected system, a WHOLE_POINTER will occupy less space than a PART_POINTER. This is why the TDF SHAPE system distinguishes them.

Cross-reference: generate §2.3.3.1, whole_to_part §2.3.3.2, Tokenisation §1.5.3

2.3.2.1.3 SHAKY_WHOLE_POINTER SHAPEs

A SHAKY_WHOLE_POINTER is a POINTER which points to the whole of an allocated space. Unlike a WHOLE_POINTER, which can cleared by a garbage collection only when all copies of it have been discarded, it can be a cleared by a garbage collection if all the other extant POINTERs to its space are also SHAKY. SHAKY POINTERs (including SHAKY_PART_POINTERs) do not have the power to preserve the piece of memory to which they point during a garbage collection. If a SHAKY POINTER points to a piece of memory all or part of which is also pointed to by a non-SHAKY POINTER, however, that piece of memory will be preserved during a garbage collection and the SHAKY POINTER will not be cleared.

A SHAKY_WHOLE_POINTER(X) can be created by applying *shake* to a WHOLE_POINTER(X).

SHAKY_WHOLE_POINTERs have equal representation if and only if they are identical - ie. they are copies of a value produced from one particular evaluation of *generate*.

The only operation (apart from data transfer) defined on SHAKY POINTERs is *firm*. Applied to a SHAKY POINTER which has not been cleared by a garbage collection, *firm* delivers the original POINTER from which the SHAKY POINTER was made. However, if the SHAKY POINTER has been cleared by a garbage collection, an absent_shaky error is produced.

Cross-reference: Garbage Collection: Discussion §2.3.2.1.1, WHOLE_POINTER SHAPEs §2.3.2.1.2, shake §2.3.3.12, generate §2.3.3.1, absent_shaky §2.2.3.2.3.1.6

2.3.2.1.4 PART_POINTER SHAPEs

A PART_POINTER is a pointer which does not necessarily point to the whole of an allocated space. Such a POINTER is created either by applying *whole_to_part* to a WHOLE_POINTER, or in the course of the constructs, *variable* and *variable_no_init* when evaluated as part of a TRACED_PROC, when a PART_POINTER is created and bound to a TAG.

Equality of representation for PART_POINTERs is defined if and only if both POINTERs are proper POINTERs (see §2.1.3.4.1.4). If both POINTERs are derived from different original POINTERs, then the representations are unequal. If both are derived from the same original POINTER, then they are equal if and only if *subtract_pointers* would give a zero SIZE.

PART_POINTERs derived from variable declarations have a limited lifetime.

Cross-reference: whole_to_part §2.3.3.2, WHOLE_POINTER SHAPEs §2.3.2.1.2, variable §2.1.3.1.4, variable_no_init §2.1.3.1.5, proper POINTERs §2.1.3.4.1.4, original POINTERs §2.1.3.4.1.3, Lifetimes: Discussion §2.1.3.12, subtract_pointers §2.1.3.4.14

2.3.2.1.5 SHAKY_PART_POINTER SHAPEs

A SHAKY_PART_POINTER is a POINTER which does not necessarily point to the whole of an allocated space. Unlike a PART_POINTER, however, it can be cleared by a garbage collection. A SHAKY_PART_POINTER(X) can be created by applying *shake* to a PART_POINTER(X).

Equality of representation for SHAKY_PART_POINTERs is defined if and only if both POINTERs are proper POINTERs (see §2.1.3.4.1.4). If both POINTERs are derived from different original POINTERs, then the representations are unequal. If both are derived from the same original POINTER, then they are equal if and only if *subtract_pointers* would give a zero size.

As with SHAKY_WHOLE_POINTERs, the only operation (apart from data transfer) which can be applied to a SHAKY_PART_POINTER is *firm*.

Cross-reference: Garbage Collection: Discussion §2.3.2.1.1, PART_POINTER SHAPEs §2.3.2.1.4, shake §2.3.3.12, subtract_pointers §2.1.3.4.14, firm §2.3.3.13

2.3.2.2 TRACED_PROC

The SHAPE describing procedure values in a garbage collecting system which reside on the garbage collected heap. The only ultimate use that can be made of a TRACED_PROC is to apply it to a parameter.

Equality of representation is undefined for TRACED_PROCs.

Cross-reference: Garbage Collection: Discussion §2.3.2.1.1, Lifetimes: Discussion §2.1.3.12, Procedures §2.1.3.6

2.3.2.3 the SHAPE UNIQUE_VAL

The SHAPE describing values which are different from every other value of the same SHAPE previously generated on the current machine, and from every UNIQUE_VAL generated on any other machine.

UNIQUE_VAL values are created by the TDF construct *make_a_new_unique_value*.

UNIQUE_VAL values are equal if and only if they are identical - ie. they are copies of a value produced from the same evaluation of *make_a_new_unique_value*.

Cross-reference: make_a_new_unique_value §2.3.3.7

2.3.2.4 unlimited integer VARIETYs

Two VARIETYs, used to represent integers of unbounded size, do not occur in Levels 0 or 1. The operations on such unbounded integers cannot cause overflow errors. The unlimited VARIETYs are:

SIGNED_UNLIMITED
UNSIGNED_UNLIMITED

SIGNED_UNLIMITED is a representation of any integer, without bound. UNSIGNED_UNLIMITED is a representation of any non-negative integer, without upper bound. Such integers can only be used freely in full (garbage collecting) TDF systems.

In the case of unlimited integers the *impossible* error handler will be chosen for the overflow error handler in arithmetic operations.

Equality of representation is undefined for unlimited integers. However the integer_test operations can be used for unlimited integers.

The LUB of a limited and an unlimited VARIETY is *top*.

2.3.3 Level 2 EXPs

All the EXPs of the previous sections shall be implemented, plus the following.

2.3.3.1 generate

arg: EXP SIZE(X) {X need not be SOME-free}

-> EXP WHOLE_POINTER(X)

arg is evaluated to produce a SIZE(X) value. Space is generated to hold a value of this SIZE and a WHOLE_POINTER(X) which points to this space is delivered as the result. The space is not initialised. If no memory is available, a signal *store_full* is produced.

The result shares with no other POINTER existing at the completion of the operation. It can never share with a POINTER produced by *variable* or *variable_no_init*, nor with any POINTER produced from such a POINTER by *add_to_ptr*, *subtract_from_ptr*, *index* or *part_field* or by any combination of these operations. The result is an original POINTER.

The result has an unbounded lifetime.

If *X* is a compound SHAPE, such as a TUPLE, then the space to hold it may contain areas of padding in order to conform to an architecture's alignment rules. This causes no difficulty if the whole TUPLE is subsequently assigned to. But if the individual fields are separately assigned to, then it is possible that no values may be put into the padding areas. But equality of TUPLEs is defined to be equality of the components, and is very likely to be implemented by comparison of all the TUPLE, including the padding. Translators therefore need to make sure that such padding has a standard vaue. Thus some clearing operation on the space generated may be necessary.

An initialised generate operation, *pack* (§2.3.3.11), is available.

Cross-reference: Pointers: Discussion §2.1.3.4.1, add_to_ptr §2.1.3.4.2, Signals: Discussion §2.1.3.11, index §2.2.3.2.17, Lifetimes: Discussion §2.1.3.12, original POINTERs §2.1.3.4.1.3, part_field §2.1.3.4.4, sharing §2.1.3.4.1.1, SHAPEs SIZEs and OFFSETs §1.5.1, subtract_from_ptr §2.1.3.4.3, variable §2.1.3.1.4, variable_no_init §2.1.3.1.5, pack §2.3.3.11, null POINTERs §2.1.3.4.1.2, store_full §2.2.3.2.3.1.3

2.3.3.2 whole_to_part

arg: EXP WHOLE_POINTER(X)

-> EXP PART_POINTER(X)

arg is evaluated to produce a WHOLE_POINTER *p*. A PART_POINTER pointing to the same space as *p* is created and delivered. If *p* is null, the result will be a null PART_POINTER.

The result will share with *p*. It has an unbounded lifetime.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, sharing §2.1.3.4.1.1, Lifetimes: Discussion §2.1.3.12, generate §2.3.3.1

2.3.3.3 make_null_whole_pointer

sh: SHAPE

-> EXP WHOLE_POINTER(*sh*)

A null WHOLE_POINTER(*sh*) is delivered. Its lifetime is unbounded.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2

2.3.3.4 make_null_part_pointer

sh: SHAPE

-> EXP PART_POINTER(*sh*)

A null PART_POINTER(*sh*) is created and delivered. Its lifetime is unbounded.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2

2.3.3.5 replace_field

component: NAT,
tuple: EXP TUPLE $(\Pi_{i=1}^{n}) P_i$, $\{n \geq 1\}$ $\{1 <= \text{component} <= n\}$
replace_by: EXP $P_{\text{component}}$

-> EXP TUPLE $\Pi_{i=1}^{n} P_i$

tuple is evaluated to produce a tuple value *t* and *replace_by* to produce a value *r*. The

value *r* will have the same SHAPE as the *component-th* element of *t*. A new tuple is created and delivered which is the same as *t* except that its *component-th* field is *r*.

Cross-reference: TUPLE §1.5.1.1, Lifetimes: Discussion §2.1.3.12, SHAPEs SIZEs and OFFSETs §1.5.1

2.3.3.6 size_of_contents

sh: SHAPE, {sh need not be SOME-free}
arg: EXP POINTER(X)

-> EXP SIZE(X)

arg is evaluated to produce a POINTER, which is either a WHOLE_POINTER, *p*, or is equal to the result of applying *whole_to_part* to a WHOLE_POINTER, *p*. If *p* was produced by *generate*, then the result is the SIZE parameter with which it was *generated*. If *p* was produced by *pack*, the result is the SIZE of the data which was *packed*.

Cross-reference: SHAPEs SIZEs and OFFSETs §1.5.1, WHOLE_POINTER SHAPEs §2.3.2.1.2, generate §2.3.3.1, pack §2.3.3.11

2.3.3.7 make_a_new_unique_value

-> EXP UNIQUE_VAL

A new UNIQUE_VAL value, different from any existing value on this or any other machine, is created and delivered. This operation is only permitted if the computer on which the program is running has the right to issue UNIQUE_VAL values. Otherwise, the effect is undefined.

Cross-reference: the SHAPE UNIQUE_VAL §2.3.2.3

2.3.3.8 make_unique_val

u:UNIQUE

-> EXP UNIQUE_VAL

A UNIQUE_VAL equal to the UNIQUE *u* is created and delivered.

Cross-reference: UNIQUE §2.1.1.8, the SHAPE UNIQUE_VAL §2.3.2.3

2.3.3.9 floor

ov_err: ERROR_TREATMENT,
v: VARIETY,
arg: EXP FLOAT(F)

-> EXP INTEGER(v)

arg is evaluated to produce a value, *a*. An integer value is created and delivered which is the largest integer not greater than *a*; the fractional part is discarded.

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

If *ov_err* is *ignore* and the VARIETY *v* is unsigned, the operation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, floating point SHAPEs §2.1.2.3.2

2.3.3.10 ceiling

ov_err: ERROR_TREATMENT,
v: VARIETY,
arg: EXP FLOAT(F)

-> EXP INTEGER(v)

arg is evaluated to produce a value, *a*. An integer value is created and delivered which is the smallest integer greater than or equal to *a*.

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

If *ov_err* is *ignore* and the VARIETY *v* is unsigned, the operation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY is signed, the effect of overflow is undefined.

Cross-reference: ERROR_TREATMENT §2.1.1.12, integer SHAPEs §2.1.2.3.1, floating point SHAPEs §2.1.2.3.2

2.3.3.11 pack

arg: EXP X {X must be SOME-free}

-> EXP WHOLE_POINTER(X)

arg is evaluated to produce a value, *a*. Enough space is generated to hold *a* and *a* is copied into it. A WHOLE_POINTER pointing to the space is created and delivered as the result.

The lifetime of the result is unbounded.

The result shares with no other POINTER existing at the completion of the operation. It can never share with a POINTER produced by *variable* or *variable_no_init*, nor with any POINTER produced from such a POINTER by *add_to_ptr*, *subtract_from_ptr*, *index* or *part_field* or by any combination of these operations. The result is an original POINTER.

This operation is similar to *generate* except that it permits an initial value to be supplied.

Cross-reference: Pointers: Discussion §2.1.3.4.1, add_to_ptr §2.1.3.4.2, Signals: Discussion §2.1.3.11, index §2.2.3.2.17, Lifetimes: Discussion §2.1.3.12, original POINTERs §2.1.3.4.1.3, part_field §2.1.3.4.4, sharing §2.1.3.4.1.1, subtract_from_ptr §2.1.3.4.3, variable §2.1.3.1.4, variable_no_init §2.1.3.1.5, generate §2.3.3.1, store_full §2.2.3.2 3.1.3, null POINTERs §2.1.3.4.1.2

2.3.3.12 shake

ptr: EXP POINTER(X)

-> EXP SHAKY_POINTER(X)

ptr is evaluated to produce a POINTER value *p*. A shaky POINTER pointing to the same space as *p* is created and delivered. If *p* is a WHOLE_POINTER, the result is a shaky WHOLE_POINTER. If the *p* is a PART_POINTER, the result is a shaky PART_POINTER. *p* will not be a null POINTER.

The result will share with *p*.

Cross-reference: Pointers: Discussion §2.1.3.4.1, null POINTERs §2.1.3.4.1.2, sharing §2.1.3.4.1.1, Lifetimes: Discussion §2.1.3 12

2.3.3.13 firm

absent: ERROR_TREATMENT,
sh_ptr: EXP SHAKY POINTER(X)

-> EXP POINTER(X)

sh_ptr will deliver a shaky POINTER value *sp*. If this shaky POINTER has not been cleared, a POINTER pointing to the same space is created and delivered as the result. If *sp* is a SHAKY_WHOLE_POINTER the result is a WHOLE_POINTER. If *sp* is a SHAKY_PART_POINTER, the result is a PART_POINTER. If *sp* is null, then the result is a null POINTER. If *sp* has been cleared, an absent_shaky error is produced and handled as specified by *absent*.

If *absent* is *ignore* its effect is undefined.

If *sp* is present, it will share with the result.

Cross-reference: Pointers: Discussion §2.1.3 4.1, null POINTERs §2.1.3.4.1.2, ERROR_TREATMENT §2.1.1.12, sharing §2.1.3.4.1.1, Lifetimes: Discussion §2.1.3.12

2.3.3.14 make_traced_procedure

param_shape: SHAPE, {param_shape will be SOME-free}
param: TAG PART_POINTER(param_shape),
body: EXP X

-> EXP TRACED_PROC

Evaluation of *make_traced_procedure* delivers an TRACED_PROC. When this procedure is applied to a parameter using *apply_proc*, space is allocated to hold a value of SHAPE *param_shape*. The value produced by the parameter, which will be of the correct SHAPE, is used to initialise it. *body* is evaluated. During the evaluation, *param* is bound to an original PART_POINTER pointing to the space. This means that evaluation of *obtain_tag(param)* will produce that POINTER. The value produced by *body* is delivered as the result of the *apply_proc* construct.

The TAG used for *param* will not be re-used. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.1.3.1.1 for a discussion of this point.

TAGs other than *param* which are used in *body* but not declared within it are called non-local TAGs. If and when the procedure is applied and its *body* evaluated, these TAGs obey the same bindings that obtained when the procedure was constructed.

The lifetime of the procedure value is the intersection of the evaluations of the bodies of all the declarations of its non-local TAGs.

If a programming language permits more than one parameter, the compiler to TDF will use *make_traced_procedure* to construct a TDF procedure whose *param_shape* is ALIGNED_TUPLE (..) (see §2.1.2.3.5). The elements will usually be identified (§2.1.3.1.3).

Cross-reference: TRACED_PROC §2.3.2.2, Lifetimes: Discussion §2.1.3.12, Binding: Discussion §2.1.3.1.1, Exceptions: Discussion §2.2.3.2.3, apply_proc §2.1.3.6.6, ALIGNED_TUPLE SHAPEs §2.1.2.3.5, identify §2.1.3.1.3

2.3.3.15 make_null_traced_procedure

-> EXP TRACED_PROC

A null TRACED_PROC is created and delivered. If this PROC is applied, the effect is undefined. The null TRACED_PROC may be tested for using *proc_is_null* or *proc_not_null*.

Cross-reference: TRACED_PROC §2.3.2.2, apply_proc §2.1.3.6.6, proc_is_null §2.1.3.6.4, proc_not_null §2.1.3.6.5

3 TDF: Optimisations

One of the requirements which TDF has been designed to satisfy is that it should contain enough information for all the normal optimisations to be performed in the target machine. TDF satisfies this requirement.

If optimisations are done before transmission, the resources of a presumably more powerful computer are available. The compiler has to be trusted to perform these optimisations correctly.

If the optimisations are done after transmission, the effects of the modifications made during installation on the target can be allowed for. The translator has to be trusted to perform these optimisations correctly.

Since compilers and translators are to be usable in any combination, it will be necessary to make a public decision about which optimisations are expected if the expected efficiencies are to be achieved.

Regardless of this decision, some TDF optimisations should be performed by every translator to which they are relevant. Every TDF translator should produce the best code it can for these cases. These optimisations are listed here. They are not concerned with such matters as register allocation, pipeline control or removing redundant jumps, which should also be performed if relevant.

All compiler writers may assume that these optimisations will be performed if they are relevant. This information is important to compiler writers in considering their TDF producing strategy.

3.1 Evaluation of Constants and Conditional Compilation

For every operation, if the expression arguments are explicit constants and the result can be expressed as an explicit constant, then it should be so expressed. The result should then be available as an explicit constant for similar consideration by any enclosing operation. If the arguments deliver constant results but the evaluation has side-effects, and if the result can be expressed as an explicit constant with side-effects, this should be done. Again, the result should be available for enclosing operations.

Every operation whose EXP arguments are explicit constants, which has an ERROR_HANDLER argument which is a LABEL, and whose effect is completely defined by its arguments should be processed as follows. If control certainly passes to the LABEL, the operation should be replaced by a suitable *goto* operation together with any necessary side-effects. Since this has a *bottom* result, no further operations after this *goto* need be translated. If control certainly does not pass to the LABEL, the operation should be replaced by one with an *impossible* error handler. In this case the LABEL is not used by the operation.

In any case where a *bottom* is produced, no subsequent unreachable code should be produced.

In a *conditional*, *repeat* or *labelled* construct, any labelled expression which is unreachable because its LABEL is not used should not be translated. This may well be because of a combination of the rules above.

In a *case* construct, if the controlling integer is an explicit constant, only the selected branch should be translated.

These rules have a bearing on conditional translation. This needs to be done in the translator, because it may be the installation process on the target which makes expressions constant or unreachable.

Cross-reference: ERROR_TREATMENT §2.1.1.12, bottom §2.1.2.1.1, labelled §2.1.3.5.7, conditional §2.1.3.5.5, repeat §2.1.3.5.6, case §2.1.3.5.4

3.2 Operations with Some Constant Arguments

Operations, only some of whose arguments are explicit constants, can also be optimised. For example, *test_eq* with one argument a constant (especially if it is zero) may be optimisable. These optimisations are expected.

3.3 Increment etc.

Expressions involving *assign* and binary or unary operators, of forms

a := (contents a) binop b

a := b binop (contents a)

a := b binop c

a := unop (contents a)

a := binop b

should be examined with a view to making best use of three-address instruction, add-to-store, increment etc. whatever the form of *a*, *b* and *c*. The choice has been made that it is better for such formations to appear 'longhand' in TDF, rather than be catered for by special operations (such as the ++ of C), because the cases are somewhat dependent on the nature of the target machine.

Cross-reference: assign §2.1.3.4.5, contents §2.1.3.4.6

3.4 Contents of Variables

Addresses of variables should only be manipulated where these addresses are themselves the required data. In common machines *contents(var)* should involve only an operand access to the variable, not the loading of the address on the stack.

3.5 Tail Recursion and Last Call

Translators should recognise a use of *apply_proc* which is the last operation in a procedure, even if this is inside such constructs as *conditional*. The use of *obtain_current_procedure* will make it possible to recognise tail recursion in this situation, and so compilers should use it whenever possible. Translators should optimise tail recursion and are encouraged to optimise last call. Care is needed to avoid these optimisations if a LABEL_VALUE is created in the procedure.

Cross-reference; LABEL_VALUE, obtain_current_procedure, apply_proc

3.6 Field Selection

When possible, a combination of field selections from a tuple or field selections from a POINTER to a TUPLE, as in *a.b.c*, should be combined. Often it will be possible to make such an access into a single displacement-from-register operand.

No code is usually necessary for a part_field selection from a variable.

Cross-reference: field §2.1.3.9.3, part_field §2.1.3.4.4

3.7 NTEST, and, test_eq etc.

The complex of optimisations involved in tests of relations, and the *and*, *or* and *not* of such tests should be optimised in the usual way. Note that a test for true has the form of *test_eq* on a 1-bit value against the constant *true*.

Cross-reference: test_eq §2.1.3.10.2, test_neq §2.1.3.10.4, true §2.2.3.2.10, false §2.2.3.2.11, and §2.1.3.8.4, or §2.1.3.8.5, not §2.1.3.8.7, NTEST §2.1.1.11

INDEX

A

B

C

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 91005	OCTOBER	1990

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
TDF SPECIFICATION

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
FOSTER, J M; BRANDRETH, M; CORE, P W; CURRIE, I F; PEELING, N E	127

Abstract

This report is primarily intended to serve as a reference manual which precisely defines the meaning of TDF. An extended introduction is included which explains the purpose of TDF and gives an overview of its structure. It also explains those concepts within the definition which may be new to the reader and defines terminology used in the rest of the document.

Some discussion sections are also included amongst tne definitions of TDF constructs. These can be read independently of the definitions and are intended to provide the reader with a broader perspective of particular areas of TDF.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED