A Guide to the TDF Specification

Issue 2.1.0 (June 1993)

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

Notice to Readers

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

Requests for information about TDF should be directed to:

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

Tel. +44 684 895314
Fax +44 684 894303

Internet peeling%hermes.mod.uk@relay.mod.uk

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

June 1993

CONTENTS

1 Introduction

This memo is intended to be a fairly detailed commentary on the specification of TDF, a kind of Talmud to the Torah. If it conflicts with the specification document, it is wrong. The aim is elucidate the various constructions of TDF, giving examples of usages both from the point of view of a producer of TDF and how it is used to construct programs on particular platforms using various installers or translators. In addition, some attempt is made to give the reasons why the particular constructions have been chosen. Most of the commentary is a distillation of questions and answers raised by people trying to learn TDF from the specification document.

Throughout this document, references like (S5.14) are headings in the TDF specification, Issue 2.1 (March 93). I use the term “compiling” or “producing” to mean the production of TDF from some source language and “translating” to mean making a program for some specific platform from TDF.

2 SORTs and TOKENs

In the syntax of language like C or Pascal, we find various syntactic units like <Expression>, <Identifier> etc. A SORT bears the same relation to TDF as these syntactic units bear to the language; roughly speaking, the syntactic unit <Expression> corresponds to the SORT EXP (S5.14) and <Identifier> to TAG (S5.30). However, instead of using BNF to compose syntactic units from others, TDF uses explicit constructors to compose its SORTs; each constructor uses other pieces of TDF of specified SORTs to make a piece of its result SORT. For example, the constructor **plus** (S5.14.71) uses an ERROR_TREATMENT and two EXPs to make another EXP.

At the moment, there are 53 different SORTS, from ACCESS (S5.1) to VARIETY (S5.53) given in tables 1 and 2. Some of these have familiar analogues in standard language construction as with EXP and TAG above. Others will be less familiar since TDF must concern itself with issues not normally addressed in language definitions. For example, the process of linking together TDF programs is at the root of the architecture neutrality of TDF and so must form an integral part of its definition. On the other hand, TDF is not meant to be a language readily accessible to the human reader or writer; computers handle it much more easily. Thus a great many choices have been made in the definition which would be intolerable in a standard language definition for the human programmer but which, paradoxically enough, make it much simpler for a computer to produce and analyse TDF

The SORTs and constructors in effect form a multi-sorted algebra. There were two principal reasons for choosing this algebraic form of definition. First, it is easy to extend - a new operation on existing constructs simply requires a new constructor. Secondly, the algebraic form is highly amenable to the automatic construction of programs. Large parts of both TDF producers and TDF translators have been created by automatic transformation of the text of the specification document itself, by extracting the algebraic signature and constructing C program which can read or produce TDF. To this extent, one can regard the specification document as a formal description of the free algebra of TDF SORTs and constructors. Of course, most of the interesting parts of the definition of TDF lies in the equivalences of parts of TDF, so this formality only covers the easy bit.

Another distinction between the TDF definition and language syntactic description is that TDF is to some extent conscious of its own SORTs so that it can specify a new construction of a given SORT. The analogy in normal languages would be that one could define a new construction with new syntax and say this is an example of an <Expression>, for example; I don't know of any standard language which permits this, although those of you with a historical bent might remember Algol-N which made a valiant attempt at it. Of course, the algebraic method of description makes it much easier to specify, rather than having to give syntax to provide the syntax for the new construction in a language.

2.1 Token applications and first-class SORTs

A new construction is introduced by the SORT TOKEN; the constructors involving TOKENs allow one to give an expansion for the TOKEN in terms of other pieces of TDF, possibly including parameters. We can encapsulate a (possibly parameterised) fragment of TDF of a suitable SORT by giving it a TOKEN as identification. Not all of the SORTs are available for this kind of encapsulation - only those which have a SORTNAME constructor (from **access** (S5.29.1) to **variety** (S5.29.19). These are the "first-class" SORTs given in table 1 on page 3. Each of these have an appropriate **_apply_token** constructor (e.g. **exp_apply_token** (S5.14.1)) give the expansion.

Table 1. First class SORTs

SORT	USAGE	SORT	USAGE
ACCESS	Properties of TAGs	AL_TAG	Name for alignment
ALIGN MENT	Abstraction of data alignment	BITFIELD_ VARIETY	Gives no of bits in bit-field with sign
BOOL	**true** or **false**	ERROR_ TREATMEN T	How to handle errors in operations
EXP	Piece of TDF program, manipulating values	FLOATING_ VARIETY	Kind of floating point number
LABEL	Mark on EXP to jump to	NAT	Non-negative static number of unbounded size
NTEST	Test in comparisons	ROUNDING_ MODE	How to round floating point operations
SHAPE	Abstraction of size and representation of values	SIGNED_ NAT	Static number of unbounded size.
TAG	Name for value in run-time program	TRANSFER-MODE	Controls special contents & assignment operations
TOKEN	Installation-time function	VARIETY	Kind of integer used in run-time program

Every TOKEN has a result SORT, i.e. the SORT of its resulting expansion and before it can be expanded, one must have its parameter SORTs. Thus, you can regard a TOKEN as having a type defined by its result and parameter SORTs and the **_apply_token** as the operator which expands the encapsulation and substitutes the parameters.

However, if we look at the signature of **exp_apply_token**:

token_value: TOKEN
token_args: BITSTREAM *param_sorts(token_value)*
$\rightarrow$ EXP *x*

we are confronted by the mysterious BITSTREAM where one might expect to find the actual parameters of the TOKEN.

To explain BITSTREAMs requires a diversion into the bit-encoding of TDF. Constructors for a particular SORT are represented in a number of bits depending on the number of constructors for that SORT; the context will determine the SORT required, so no more bits are required. Thus since there is only one constructor for UNITs, no bits are required to represent **make_unit**; there are 85 different constructors for EXPs so 7 bits are required to cover all the EXPs. The parameters of each constructor have known SORTs and so their representations are just concatenated after the representation of the constructor[1]. While this is a very compact representation, it suffers from the defect that one must decode it even just to skip over it. This is very irksome is some applications, notably the TDF linker which is not interested detailed expansions. Similarly, in translators there are places where one wishes to skip over a token application without knowledge of the SORTs of its parameters. Thus a BITSTREAM is just an encoding of some TDF, preceded by the number of bits it occupies. Applications can then skip over

1. There are facilities to allow extensions to the number of constructors, so it is not quite as simple as this

BITSTREAMs trivially. Similar considerations apply to BYTESTREAMs used elsewhere; here the encoding is preceded by the number of bytes in the encoding and is aligned to a byte boundary to allow fast copying.

2.2 Token definitions

Thus the *token_args* parameter of **exp_apply_token** is just the BITSTREAM formed from the actual parameters in the sequence described by the definition of the *token_value* parameter. This will be given in a TOKEN_DEFN somewhere with constructor **token_definition** (S5.46.1):

result_sort: SORTNAME
tok_params: LIST(TOKFORMALS)
body: *result_sort*
→ TOKEN_DEFN

The *result_sort* is the SORT of the construction of *body*; e.g. if *result_sort* is formed from **exp** (S5.29.7) then *body* would be constructed using the EXP constructors and one would use **exp_apply_token** to give the expansion. The list *tok_params* gives the formal parameters of the definition in terms of TOKFORMALS constructed using **make_tok_formals**(S5.47.1):

sn: SORTNAME
tk: TDFINT
→ TOKFORMALS

The TDFINT *tk* will be the integer representation of the formal parameter expressed as a TOKEN whose result sort is *sn* (see more about name representation in section 2.1 on page 7). To use the parameter in the body of the TOKEN_DEFN, one simply uses the **_apply_token** appropriate to *sn*.Note that sn may be a TOKEN but the *result_sort* may not.

Hence the BITSTREAM *param_sorts*(*token_value*) in the actual parameter of **exp_apply_token** above is simply formed by the catenation of constructions of the SORTs given by the SORTNAMEs in the *tok_params* of the TOKEN being expanded.

Usually one gives a name to a TOKEN_DEFN using to form a TOKDEF using **make_tokdef** (S5.43.1):

tok: TDFINT
def: BITSTREAM TOKEN_DEFN
→ TOKDEF

Here, *tok* gives the name that will be used to identify the TOKEN whose expansion is given by *def*. Any use of this TOKEN (e.g. in **exp_apply_token**) will be given by **make_token**(*tok*) (S5.45.2). Once again, a BITSTREAM is used to encapsulate the TOKEN_DEFN.

One can also use a TOKEN_DEFN in an anonymous fashion by giving it as an actual parameter of a TOKEN which itself demands a TOKEN parameter. To do this one simply uses **use_tokdef** (S5.45.3):

tdef: BITSTREAM TOKEN_DEFN
→ TOKEN

2.3 A simple use of a TOKEN

The crucial use of TOKENs in TDF is to provide abstractions of APIs (see section 9 on page 34) but they are also used as shorthand for commonly occurring constructions. For example, given the TDF constructor **plus**, mentioned above, we could define a plus with only two EXP parameters more suitable to C by using the **wrap** (S5.13.4) constructor as the ERROR_TREATMENT:

```
make_tokdef (C_plus,
    token_definition(
        exp(),
        (make_tokformals(exp(), l), make_tokformals(exp(), r)),
        plus(wrap(), exp_apply_token(l, ()), exp_apply_token(r,())
    )
)
```

2.4 Second class SORTs

Second class SORTs (given in table 2 on page 6) cannot be TOKENised. These are the "syntactic units" of TDF which the user cannot extend; he can only produce them using the constructors defined in core-TDF.

Some of these constructors are implicit. For example, there are no explicit constructors for LIST or SLIST which are both used to form lists of SORTs; their construction is simply part of the encoding of TDF. However, it is forseen that LIST constructors would be highly desireable and there will probably extensions to TDF to promote LIST from a second-class SORT to a first-class one. This will not apply to SLIST or to the other SORTs which have implicit constructions. These include BITSTREAM, BYTESTREAM, TDFINT, TDFIDENT and TDFSTRING.

Table 2. Sorts without SORTNAMEs

SORT	USAGE	SORT	USAGE
AL_TAGDEF	Alignment name definition	AL_TAGDEF_PROPS	Body of UNIT containing AL_TAGDEFs
BITSTREAM	Encapsulation of a bit encoding	BYTE STREAM	Encapsulation of a byte encoding
CAPSULE	Independent piece of TDF program	CAPSULE_LINK	No and kind of linkable entities in CAPSULE
CASELIM	Bounds in **case** constructor	EXTERNAL	External name used to connect CASULE name.
EXTERN_LINK	List of LINKEXTERNs in CAPSULE	GROUP	List of UNITs with same identification.
LINK	Connects names in CAPSULE	LINK EXTERN	Used to connect CAPSULE names to outside world
LINKS	List of LINKs	LIST(AUX)	List of AUX SORTs; will have SORTNAME later
PROPS	Program info in a UNIT	SLIST(AUX)	List of AUX SORTs; will not have SORTNAME later
SORTNAME	SORT which can be parameter of TOKEN	TAGDEC	Declaration of TAG at UNIT level

Table 2. Sorts without SORTNAMEs

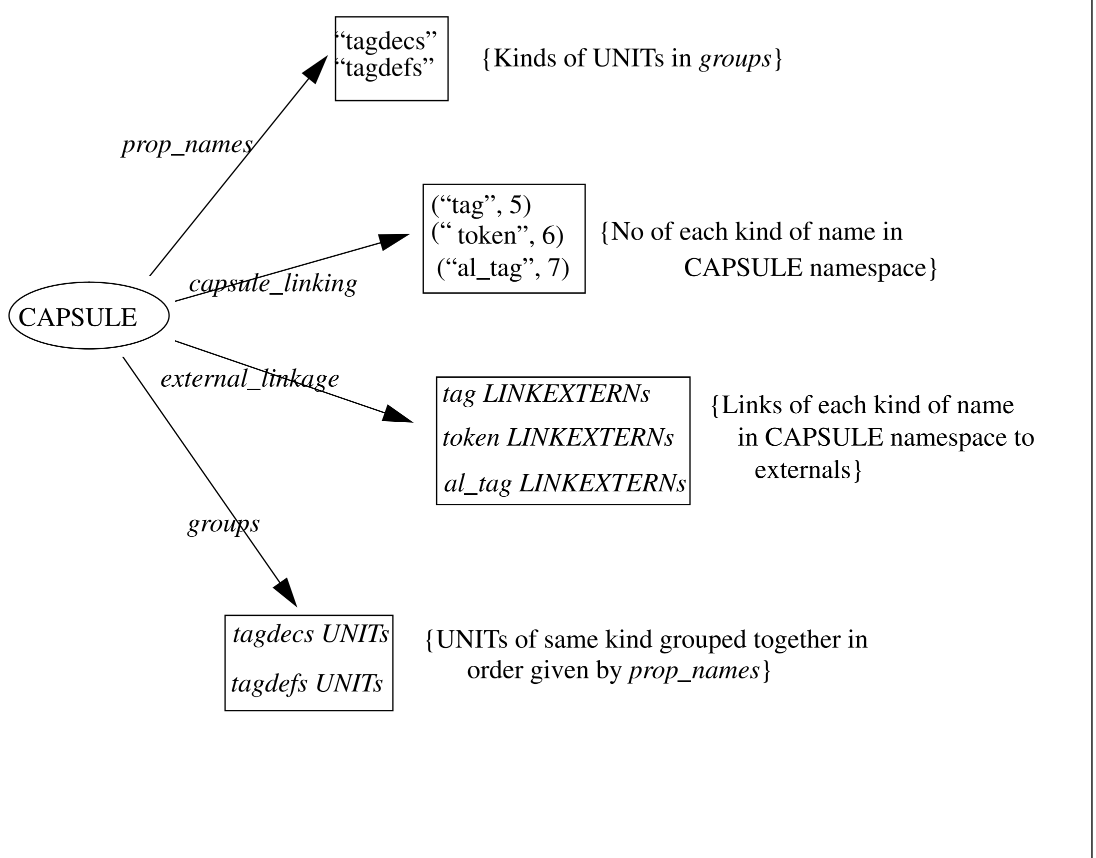
SORT	USAGE	SORT	USAGE
TAGDEC_ PROPS	Body of UNIT containing TAGDECs	TAGDEF	Definition of TAG at UNIT level
TAGDEF_ PROPS	Body of UNIT containing TAGDEFs	TAGSHACC	A formal parameter
TDFBOOL	TDF encoding for a boolean	TDFIDENT	TDF encoding of a byte string
TDFINT	TDF encoding of an integer	TDFSTRING	TDF encoding of n-bit byte string
TOKDEC	Declaration of a TOKEN	TOKDEC_ PROPS	Body of UNIT containing TOKDECs
TOKDEF	Definition of a TOKEN	TOKDEF_ PROPS	Body of UNIT containing TOKDEFs
TOKEN_ DEFN	Defines TOKEN expansion	TOKFORMA LS	Sort and name for parameters in TOKEN_DEFN
UNIQUE	World-wide name	UNIT	Component of CAPSULE with LINKs to other UNITs
VERSION	Version no of TDF	VERSION_ PROPS	Body of UNIT containing version information

3 CAPSULEs and UNITs

A CAPSULE(S5.10) is typically the result of a single compilation - one could regard it as being the TDF analogue of a Unix .o file. Just as with .o files, a set of CAPSULEs can be linked together to form another. Similarly, a CAPSULE may be translated to make program for some platform, provided certain conditions are met. One of these conditions is obviously that a translator exists for the platform, but there are others. They basically state that any names that are undefined in the CAPSULE can be supplied by the system in which it is to be run. For example, the translator could produce assembly code with external identifiers which will be supplied by some system library.

3.1 make_capsule and name-spaces

The only constructor for a CAPSULE is **make_capsule**. Its basic function is to compose together UNITs which contain the declarations and definitions of the program. The signature of **make_capsule** (S5.10.1) looks rather daunting and is probable best represented graphically.



AN EXAMPLE OF A CAPSULE

The diagram gives an example of a CAPSULE using the same components as in the following text.

Each CAPSULE has its own name-space, distinct from all other CAPSULEs' name-spaces and also from the name-spaces of its component UNITs (see section 2.1.2 on page 8). There are several different kinds of names in TDF and each name-space is further subdivided into one for each kind of name. The number of different kinds of names is potentially unlimited but only three are used in core-TDF, namely "tag", "token" and "al_tag". Those names in a "tag" name-space generally correspond to identifiers in normal programs and I shall use these as the paradigm for the properties of them all.

The actual representations of a "tag" name in a given name-space is an integer, described as SORT TDFINT. These integers are drawn from a contiguous set starting from 0 up to some limit given by the constructor which introduces the name-space. For CAPSULE name-spaces, this is given by the *capsule_linking* parameter of **make_capsule**:

capsule_linking: SLIST(CAPSULE_LINK)

In the most general case in core-TDF, there would be three entries in the list introducing limits using **make_capsule_link** (S5.11.1) for each of the "tag", "token" and "al_tag" name-spaces for the CAPSULE. Thus if:

capsule_linking = (**make_capsule_link**("tag", 5),
make_capsule_link("token", 6),
make_capsule_link("al_tag", 7))

there are 5 CAPSULE "tag" names used within the CAPSULE, namely 0, 1, 2, 3 and 4; similarly there are 6 "token" names and 7 "al_tag" names.

3.1.1 External linkages

The context of usage will always determine when and how an integer is to be interpreted as a name in a particular name-space. For example, a TAG in a UNIT is constructed by **make_tag** (S5.30.2)applied to a TDFINT which will be interpreted as a name from that UNIT's "tag" name-space. An integer representing a name in the CAPSULE name-space would be found in a LINKEXTERN of the *external_linkage* parameter of **make_capsule**.

external_linkage: SLIST(EXTERN_LINK)

Each EXTERN_LINK (S5.16) is itself formed from an SLIST of LINKEXTERNs given by **make_extern_link** (S5.16.1). The order of the EXTERN_LINKs determines which name-space one is dealing with; they are in the same order as given by the *capsule_linkage* parameter. Thus, with the *capsule_linkage* given above, the first EXTERN_LINK would deal with the "tag" name-space; Each of its component LINKEXTERNs (S5.21) constructed by **make_linkextern** would be identifying a tag number with some name external to the CAPSULE; for example one might be:

make_linkextern (4, **string_extern**("printf"))

This would mean: identify the CAPSULE's "tag" 4 with an name called "printf", external to the module. The name "printf" would be used to linkage external to the CAPSULE; any name required outside the CAPSULE would have to be linked like this.

3.1.2 UNITs

This name "printf", of course, does not necessarily mean the C procedure in the system library. This depends both on the system context in which the CAPSULE is translated and also the meaning of the CAPSULE "tag" name 4 given by the component UNITs of the CAPSULE in the *groups* parameter of **make_capsule**:

groups: SLIST(GROUP)

Each GROUP (S5.18) in the *groups* SLIST will be formed by sets of UNITs of the same kind. Once again, there are a potentially unlimited number of kinds of UNITs but core-TDF only uses those named "al_tagdefs", "tagdecs", "tagdefs", "tokdecs" and "tokdefs"[1]. These names will appear (in the same order as in *groups*) in the *prop_names* parameter of **make_capsule**, one for each kind of UNIT appearing in the CAPSULE:

prop_names: SLIST*(*TDFIDENT)

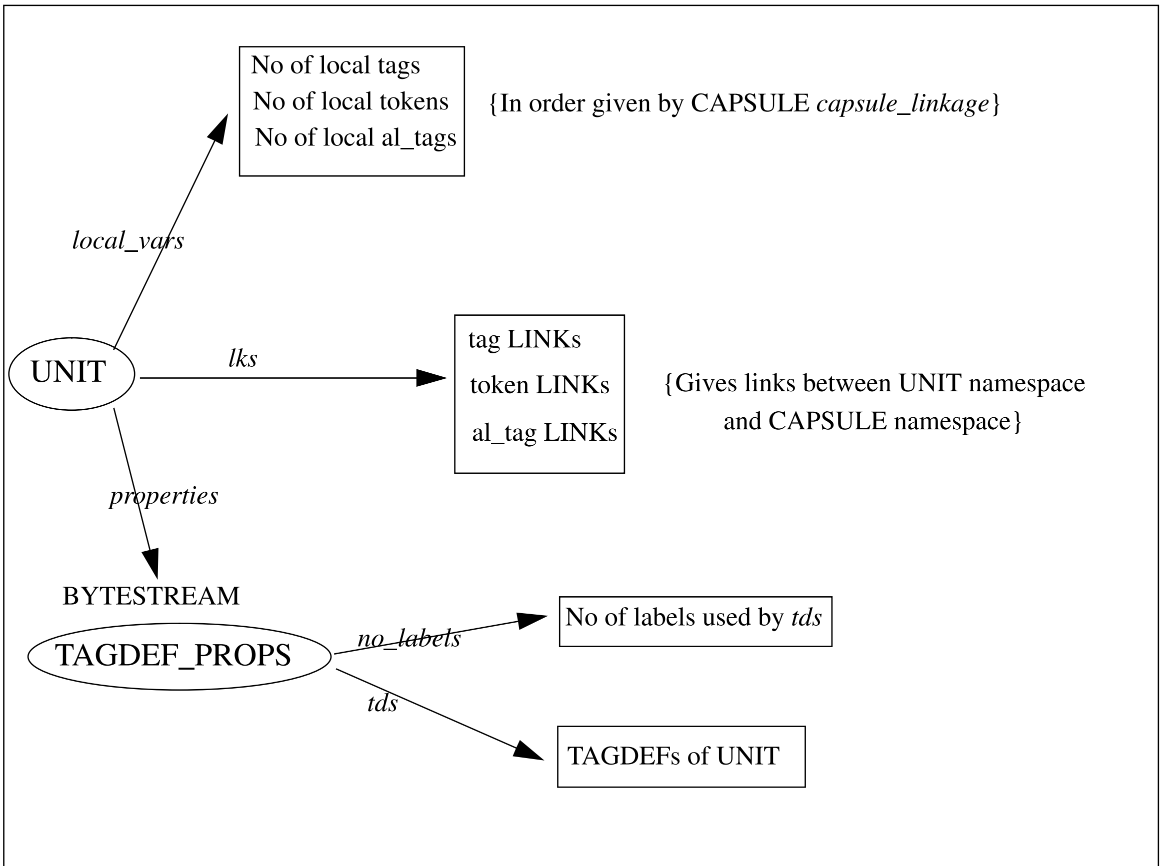
Thus if:

prop_names = ("tagdecs", "tagdefs")

then, the first element of *groups* would contain only "tagdecs" UNITs and and the second would contain only "tagdefs" UNITs. A "tagdecs" UNIT contains things rather like a set of global identifier declarations in C, while a "tagdefs" UNIT is like a set of global definitions of identifiers.

3.1.3 make_unit

Now we come to the construction of UNITs using **make_unit** (S5.50.1), as in the diagram below



AN EXAMPLE OF A TAGDEF UNIT

First we give the limits of the various name-spaces local to the UNIT in the *local_vars* parameter:

local_vars: SLIST(TDFINT*)*

Just in the same way as with *external_linkage*, the numbers in local_vars correspond (in the same order) to the spaces indicated in *capsule_linking* in section 2.1 on page 7. With our example,the first element of *local_vars* gives the number of "tag" names local to the UNIT, the second gives the number of "token" names local to the UNIT etc. These will include **all** the names used in the body of the UNIT. Each declaration of a TAG, for example, will use a new number from the "tag" name-space; there is no hiding or reuse of names within a UNIT.

1. The C producer also makes "tld2" UNITs (which gives usage information for TAGs to aid the linker, tld, to discover which TAGs have definitions in libraries) and, optionally, "diagnostics" UNITs (to give run-time diagnostic information).

3.1.4 LINK

Connections between the CAPSULE name-spaces and the UNIT name-spaces are made by LINKs (S5.20) in the *lks* parameter of **make_unit**:

lks: SLIST(LINKS*)*

Once again, *lks* is effectively indexed by the kind of name-space a. Each LINKS (S5.22) is an SLIST of LINKs each of which which establish an identity between names in the CAPSULE name-space and names in the UNIT name-space. Thus if the first element of *lks* contains:

make_link(42, 4)

then, the UNIT "tag" 42 is identical to the CAPSULE "tag" 4.

Note that names from the CAPSULE name-space only arise in two places, LINKs and LINK_EXTERNs. Every other use of names are derived from some UNIT name-space.

3.2 Definitions and declarations

The encoding in the *properties*:BYTSTREAM parameter of a UNIT is a PROPS (S5.25), for which there are five constructors corresponding to the kinds of UNITs in core-TDF, **make_al_tagdefs**, **make_tagdecs**, **make_tagdefs**, **make_tokdefs** and **make_tokdecs.** Each of these will declare or define names in the appropriate UNIT name-space which can be used by **make_link** in the UNIT's *lks* parameter as well as elsewhere in the *properties* parameter. The distinction between "declarations" and "definitions" is rather similar to C usage; a declaration provides the "type" of a name, while a definition gives its meaning. For tags, the "type" is the SORT SHAPE (see below) and for tokens it is the SORTNAME of the result of the TOKEN.

Taking it as a paradigm for PROPS, we have **make_tagdefs** (S5.35.1):

no_labels: TDFINT
tds: SLIST(TAGDEF)
→ TAGDEF_PROPS

The *no_labels* parameter introduces the size of yet another name-space local to the PROPS, this time for the LABELs (S5.19) used in the TAGDEFs. Each TAGDEF (S5.34) in *tds* will define a "tag" name in the UNIT's name-space. The order of these TAGDEFs is immaterial since the initialisations of the tags are constants whose values can be solved at translate time or load time. Note that are two constructors for TAGDEF, **make_id_tagdef** and **make_var_tagdef** both with the same signature:

t: TDFINT
e: EXP x
→ TAGDEF

Here *t* is tag name and *e* is its initialisation. The distinction between then arises with the use of **obtain_tag** (S5.14.59) in an EXP. If its tag parameter is derived from a **make_id_tagdef**, the result of **obtain_tag** is the value of the initialisation *e*. If it is derived from **make_var_tagdef**, its value is a pointer to space which originally contained the initialisation. In one case the tag is bound to the initial value; in the other, space is reserved to hold the initialisation and the tag is bound to a pointer to that space. There is a similar distinction between tags introduced to be locals of a procedure using **identify** and **variable** (see section 4.2.1 on page 19)

3.2.1 Scopes and linking

Only names introduced by AL_TAGDEFS, TAGDEFS, TAGDECs, TOKDECs and TOKDEFs can be used in other UNITs (and then, only via the *lks* parameters of the UNITs involved). You can regard them as being similar to C global declarations. Token definitions include their declarations implicitly; however this is not true of tags. This means that any CAPSULE which uses or defines a tag across UNITs must include a TAGDEC for that tag in its “tagdecs” UNITs. A TAGDEC (S5.32)is constructed using either **make_id_tagdec** or **make_var_tagdec**, both with the same signature:

t_intro:	TDFINT
acc:	OPTION(ACCESS)
x:	SHAPE
$\rightarrow$	TAGDEC

Here the tagname is given by *t_intro*; the SHAPE *x* will defined the amount of space required for the tag (this is analogous to the type in a C declaration). The *acc* field will define certain properties of the tag not implicit in its SHAPE; I shall return to the kinds of properties envisaged in discussing local declarations in section 4.2 on page 19.

Most program will appear in the “tagdefs” UNITs - they will include the definitions of the procedures of the program which in turn will include local definitions of tags for the locals of the procedures.

The standard TDF linker allows one to link CAPSULEs together using the name identifications given in the LINKEXTERNs, perhaps hiding some of them in the final CAPSULE. It does this just by generating a new CAPSULE name-space, grouping together component UNITs of the same kind and replacing their *lks* parameters with values derived from the new CAPSULE name-space without changing the UNITs’ name-spaces or their *props* parameters. The operation of grouping together UNITs is effectively assumed to be associative, commutative and idempotent e.g. if the same tag is declared in two capsules it is assumed to be the same thing . It also means that there is no implied order of evaluation of UNITs or of their component TAGDEFs

Different languages have different conventions for deciding how programs are actually run. For example, C requires the presence of a suitably defined “main” procedure; this is usually enforced by requiring the system ld utility to bind the name “main” along with the definitions of any library values required. Otherwise, the C conventions are met by standard TDF linking. Other languages have more stringent requirements. For example, C++ requires dynamic initialisation of globals. As the only runnable code in TDF is in procedures, C++ would require an additional linking phase to construct a “main” procedure which calls the initialisation procedures of each CAPSULE involved. A “C++init” UNIT would probably be used to used to indicate which of the procedures in the CAPSULE are to be used for initialisation.

4 SHAPEs, ALIGNMENTs and OFFSETs.

In most languages there is some notion of the type of a value. This is often an uncomfortable mix of a definition of a representation for the value and a means of choosing which operators are applicable to the value. The TDF analogue of the type of value is its SHAPE (**S3.20**). A SHAPE is only concerned with the representation of a value, being an abstraction of its size and alignment properties. Clearly an architecture-independent representation of a program cannot say, for example, that a pointer is 32 bits long; the size of pointers has to be abstracted so that translations to particular architectures can choose the size that is apposite for the platform.

4.1 Shapes

There are ten different basic constructors for the SORT SHAPE from **bitfield** (S5.27.3) to **top** (S5.27.12)as shown in table 3. SHAPEs arising from those constructors are used as qualifiers (just using an upper case version of the constructor name) to various SORTs in the definition; for example, EXP TOP is an expression with **top** SHAPE. This is just used for definitional purposes only; there is no SORT SHAPENAME as one has SORTNAME.

Table 3.

SHAPE	QUALIFIER SORT	USAGE	ALIGNMENT
BITFIELD(**v**)	BITFIELD_ VARIETY	As in C bitfields e.g. .., int x:5..	{**v**}
BOTTOM		It never gets here e.g. goto	None
COMPOUND(**sz**)	OFFSET(**x**, **y**)	Structs or unions; OFFSET **sz** is size	**x** $\supseteq$ Set-union of field alignments
FLOATING(**fv**)	FLOATING_ VARIETY	Floating point numbers	{**fv**}
INTEGER(**v**)	VARIETY	Integers, including chars	{**v**}
NOF(**n, s**)	(NAT, SHAPE)	Tuple of **n** values of SHAPE **s**	{*alignment*(**s**)}
OFFSET(**a1, a2**)	(ALIGNMENT, ALIGNMENT)	Offsets in memory; **a1** $\supseteq$ **a2**.	{*offset*}
POINTER(**a**)	ALIGNMENT	Pointers in memory	{*pointer*}
PROC		Procedure values	{*proc*}
TOP		No value; e.g. result of assign	{ }

In the TDF specification of EXPs, you will observe that all EXPs in constructor signatures are all qualified by the SHAPE name; for example, a parameter might be EXP INTEGER(v). This merely means that for the construct to be meaningful the parameter must be derived from a constructor defined to be an EXP INTEGER(v). You might be forgiven for assuming that TDF is hence strongly-typed by its SHAPEs. This is not true; the producer must get it right. There are some checks in translators, but these are not exhaustive and are more for the benefit of translator writers than for the user. A tool for testing the SHAPE correctness of a TDF program would be useful but has yet to be written.

4.1.1 TOP, BOTTOM, LUB

Two of the SHAPE constructions are rather specialised; these are TOP and BOTTOM. The result of any expression with a TOP shape will always be discarded; examples are those produced by **assign** (S5.14.6) and **integer_ test** (S5.14.34). A BOTTOM SHAPE is produced by an expression which will leave the current flow of control e.g. **goto** (S5.14.31). The significance of these SHAPEs only really impinges on the computation of the shapes of constructs which have alternative expressions as results. For example, the result of **conditional** (S5.14.15) is the result of one of its component expressions. In this case, the SHAPE of the result is described as the LUB of the SHAPEs of the components. This simply means that if one of the component SHAPEs is TOP then the resulting SHAPE is TOP; if one is BOTTOM then the resulting SHAPE is the SHAPE of the other; otherwise both component SHAPEs must be equal and is the resulting SHAPE. Since this operation is associative, commutative and idempotent, we can speak quite unambiguously of the LUB of several SHAPEs.

4.1.2 INTEGER, FLOATING, BITFIELD, PROC

Integer values in TDF have shape INTEGER(v) where v is of SORT VARIETY. The constructor for this SHAPE is **integer** (S5.27.7) with a VARIETY parameter. The basic constructor for VARIETY is **var_limits** (S5.51.3) which has a pair of signed natural numbers as parameters giving the limits of possible values that the integer can attain. The SHAPE required for a 32 bit signed integer would be:

integer(**var_limits**(-2^{31}, $2^{31}-1$))

while an unsigned char is:

integer(**var_limits**(0, 255))

A translator should represent each integer variety by an object big enough (or bigger) to contain all the possible values with limits of the VARIETY. That being said, I must confess that most current translators do not handle integers of more than 32 bits, but this will be rectified in due course.

Similarly, floating point numbers have shape FLOATING qualified by a FLOATING_VARIETY. A FLOATING_ VARIETY (S5.51) specifies the base, number of mantissa digits, and maximum and minimum exponent. Once again, it is intended that the translator will choose a representation which will contain all possible values, but in practice only those which are included in IEEE float, double and extended are actually implemented.

A number of contiguous bits have shape BITFIELD, qualified by a BITFIELD_VARIETY (**S3.4**) which gives the number of bits involved and whether these bits are to be treated as signed or unsigned integers. Current translators put a maximum of 32 on the number of bits.

The representational SHAPEs of procedure values is given by PROC with constructor **proc** (S5.27.11). I shall return to this in the description of the operations which use it.

4.1.3 Non-primitive SHAPEs

The construction of the other four SHAPEs involves either existing SHAPEs or the alignments of existing SHAPEs. These are constructed by **compound** (S5.27.5), **nof** (S5.27.8), **offset** (S5.27.9)and **pointer** (S5.27.10). Before describing these, we require a digression into what is meant by alignments and offsets.

4.2 Alignments

In most processor architectures there are limitations on how one can address particular kinds of objects in convenient ways. These limitations are usually defined as part of the ABI for the processor. For example, in the MIPs processor the fastest way to access a 32-bit integer is to ensure that the address of the integer is aligned on a 4-

byte boundary in the address space; obviously one can extract a mis-aligned integer but not in one machine instruction. Similarly, 16-bit integers should be aligned on a 2-byte boundary. In principle, each primitive object could have similar restrictions for efficient access and these restrictions could vary from platform to platform. Hence, the notion of alignment has to be abstracted to form part of the architecture independent TDF - we cannot assume that any particular alignment regime will hold universally.

The abstraction of alignments clearly has to cover compound objects as well as primitive ones like integers. For example, if a field of structure in C is to be accessed efficiently, then the alignment of the field will influence the alignment of the structure as whole; the structure itself could be a component of a larger object whose alignment must then depend on the alignment of the structure and so on. In general, we find that a compound alignment is given by the maximum alignment of its components, regardless of the form of the compound object e.g. whether it is a structure, union, array or whatever.

This gives an immediate handle on the abstraction of the alignment of a compound object - it is just the set of abstractions of the alignments of its components. Since “maximum” is associative, commutative and idempotent, the component sets can be combined using normal set-union rules. In other words, a compound alignment is abstracted as the set of alignments of the primitive objects which make up the compound object. Thus the alignment abstraction of a C structure with only float fields is the singleton set containing the alignment of a float while that of a C union of an int and this structure is a pair of the alignments of an int and a float.

4.2.1 ALIGNMENT constructors

The TDF abstraction of an alignment has SORT ALIGNMENT (S5.5). The constructor, **unite_alignments**(S5.5.5), gives the set-union of its ALIGNMENT parameters; this would correspond to taking a maximum of two real alignments in the translator.

The constructor , **alignment**(S5.5.3), gives the ALIGNMENT of a given SHAPE according to the rules given in the definition. These rules effectively *define* the primitive ALIGNMENTs as in the ALIGNMENT column of table 3. Those for PROC, all OFFSETs and all POINTERs are constants regardless of any SHAPE qualifiers. Each of the INTERGER VARIETYs, each of the FLOATING VARIETYs and each of the BITFIELD VARIETYs have their own ALIGNMENTs. These ALIGNMENTs will be bound to values apposite to the particular platform at translate-time. The ALIGNMENT of TOP is conventionally taken to be the empty set of ALIGNMENTs (corresponding to the minimum alignment on the platform).

4.2.2 Special alignments

There are four other special ALIGNMENTs. The alignment of a code address is $\{code\}$ given by **code_alignment** (S5.5.6); this will be the alignment of a pointer given by **make_local_lv** (S5.14.44) giving the value of a label. The other three special ALIGNMENTs are considered to include all of the others, but remain distinct. These are **frame_alignment** (S5.5.7) which gives the alignment of a frame-pointer delivered by **current_env** (S5.14.18); **alloca_alignment** (S5.5.8) which gives the alignment of a local pointer delivered by **local_alloc** (S5.14.37); and **var_param_alignment** (S5.5.9) which is the alignment required by a var_param of procedure definition (see section 4.1.1 on page 18)

4.2.3 AL_TAG, make_al_tagdef

Alignments can also be named as AL_TAGs using **make_al_tagdef** (S5.3.1). There is no corresponding **make_al_tagdec** since AL_TAGs are implicitly declared by their constructor, **make_al_tag** (S5.2.2). The main reason for having names for alignments is to allow one to resolve the ALIGNMENTs of recursive data structures. If, for example, we have mutually recursive structures, their ALIGNMENTs are best named and given as a set of equations formed by AL_TAGDEFs. A translator can then solve these equations trivially by substitution; this is easy because the only significant operation is set-union.

4.3 Pointer and offset SHAPEs

A pointer value must have a form which reflects the alignment of the object that it points to; for example, in the MIPs processor, the bottom two bits of a pointer to an integer must be zero. The TDF SHAPE for a pointer is

POINTER qualified by the ALIGNMENT of the object pointed to. The constructor **pointer** (S5.27.10) uses this alignment to make a POINTER SHAPE.

4.3.1 OFFSET

Expressions which give sizes or offsets in TDF have an OFFSET SHAPE. These are always described as the difference between two pointers. Since the alignments of the objects pointed to could be different, an OFFSET is qualified by these two ALIGNMENTs. Thus an EXP OFFSET(X,Y) is the difference between an EXP POINTER(X) and an EXP POINTER(Y). In order for the alignment rules to apply, the set X of alignments must include Y. The constructor **offset** (S5.27.9) uses two such alignments to make an OFFSET SHAPE. However, many instances of offsets will be produced implicitly by the offset arithmetic, e.g., **offset_pad** (S5.14.66):

a: ALIGNMENT
*arg*1: EXP OFFSET(*z, t*)
$\rightarrow$ EXP OFFSET($z \cup a$, a)

This gives the next OFFSET greater or equal to *arg1* at which an object of ALIGNMENT *a* can be placed. It should be noted that the calculation of shapes and alignments are all translate-time activities; only EXPs should produce runnable code. This code, of course, may depend on the shapes and alignments involved; for example, **offset_pad** might round up *arg1* to be a multiple of four bytes if *a* was an integer ALIGNMENT and *z* was a character ALIGNMENT. Translators also do extensive constant analysis, so if *arg1* was a constant offset, then the round-off would be done at translate-time to produce another constant.

.

4.4 Compound SHAPEs

The alignments of compound SHAPEs (i.e. those arising from the constructors **compound** and **nof**) are derived from the constructions which produced the SHAPE. To take the easy one first, the constructor **nof** has signature:

n: NAT
s: SHAPE
$\rightarrow$ SHAPE

This SHAPE describes an array of *n* values all of SHAPE *s*; note that *n* is a natural number and hence is a constant known to the producer. Throughout the definition this is referred to as the SHAPE NOF(n, s). The ALIGNMENT of such a value is **alignment**(s); i.e. the alignment of an array is just the alignment of its elements.

The other compound SHAPEs are produced using **compound**:

sz: EXP OFFSET(*x, {}*)
$\rightarrow$ SHAPE

The *sz* parameter gives the minimum size which can accommodate the SHAPE.

4.4.1 Offset arithmetic with compound shapes

The constructors **offset_add** (S5.14.60), **offset_zero** (S5.14.69) and **shape_offset** (S5.14.80) are used together with **offset_pad** to implement (*inter alia*) selection from structures represented by COMPOUND SHAPEs. Starting from the zero OFFSET given by **offset_zero**, one can construct an EXP which is the offset of a field by padding and adding offsets until the required field is reached. The value of the field required could then be extracted using **component** (S5.14.13) or **add_to_ptr** (S5.14.3). Most producers would define a TOKEN for the EXP OFFSET of each field of a structure or union used in the program simply to reduce the size of the TDF

The SHAPE of a C structure consisting of an char followed by an int would require *x* to be the set consisting of two INTEGER VARIETYs, one for int and one for char, and *sz* would probably have been constructed like:

sz = **offset_add**(**offset_pad**(int_al, **shape_offset**(char)), **shape_offset**(int))

The various rules for the ALIGNMENT qualifiers of the OFFSETs give the required SHAPE; these rules also ensure that offset arithmetic can be implemented simply using integer arithmetic for standard architectures (see section 12.1 on page 44). Note that the OFFSET computed here is the minimum size for the SHAPE. This would not in general be the same as the difference between successive elements of an array of these structures which would have SHAPE OFFSET(*x*, *x*) as produced by **offset_pad**(*x*, *sz*). For examples of the use of OFFSETs to access and create structures, see section 11 on page 42.

4.4.2 offset_mult

In C, all structures have size known at translate-time. This means that OFFSETs for all field selections of structures and unions are translate-time constants; there is never any need to produce code to compute these sizes and offsets. Other languages (notably Ada) do have variable size structures and so sizes and offsets within these structures may have to be computed dynamically. Indexing in C will require the computation of dynamic OFFSETs; this would usually be done by using **offset_mult** (S5.14.64) to multiply an offset expression representing the stride by an integer expression giving the index:

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

and using **add_to_ptr** with a pointer expression giving the base of the array with the resulting OFFSET.

4.4.3 OFFSET ordering and representation

There is an ordering defined on OFFSETs with the same alignment qualifiers, as given by **offset_test** (S5.14.68) and **offset_max** (S5.14.68) having properties like:

shape_offset(S) ≥ **offset_zero**(**alignment**(S))
A ≥ B iff **offset_max**(A,B) = A
offset_add(A, B) ≥ Awhere B ≥ **offset_zero**(some compatible alignment)

In most machines, OFFSETs would be represented as single integer values with the OFFSET ordering corresponding to simple integer ordering. The **offset_add** constructor just translates to simple addition with **offset_zero** as 0 with similar correspondences for the other offset constructors. You might well ask why TDF does not simply use integers for offsets, instead of introducing the rather complex OFFSET SHAPE. The reasons are twofold. First, following the OFFSET arithmetic rules concerned with the ALIGNMENT qualifiers will ensure that one never extracts a value from a pointer with the wrong alignment by, for example, applying **contents** to an **add_to_pointer**. This frees TDF from having to define the effect of strange operations like forming a float by taking the contents of a pointer to a character which may be mis-aligned with respect to floats - a heavy operation on most processors. The second reason is quite simple; there are machines which cannot represent OFFSETs by a single integer value.

The iAPX-432 is a fairly extreme example of such a machine; it is a “capability” machine which must segregate pointer values and non-pointer values into different spaces. On this machine a value of SHAPE POINTER({-*pointer*, int}) (e.g. a pointer to a structure containing both integers and pointers) could have two components[1]; one referring to the pointers and another to the integers. In general, offsets from this pointer would also have two components, one to pick out any pointer values and the other the integer values. This would obviously be the case if the original POINTER referred to an array of structures containing both pointers and integers; an offset to an element of the array would have SHAPE OFFSET({*pointer*, int},{*pointer*, int}); both elements of the offset would have to be used as displacements to the corresponding elements of the pointer to extract the structure element. The OFFSET ordering is now given by the comparison of both displacements. Using this method, one finds that pointers in store to non-pointer alignments are two words in different blocks and pointers to pointer-alignments are four words, two in one block and two in another. This sounds a very unwieldy machine compared to normal machines with linear addressing. However, who knows what similar strange machines will appear in future; the basic conflicts between security, integrity and flexibility that the iAPX-432 sought to resolve are still with us. For more on the modelling of pointers and offsets see section 12 on page 44.

1. In fact, I believe that most compilers on the iAPX-432 represent pointers as integer displacements into a single pointer area - unfortunately, obviating most of the claimed advantages of this kind of capability machine.

4.5 BITFIELD alignments

Even in standard machines, one finds that the size of a pointer may depend on the alignment of the data pointed at. Most machines do not allow one to construct pointers to bits with the same facility as other alignments. This usually means that pointers in memory to BITFIELD VARIETYs must be implemented as two words with an address and bit displacement. One might imagine that a translator could implement BITFIELD alignments so that they are the same as the smallest natural alignment of the machine and avoid the bit displacement, but this is not the intention of the definition. On any machine for which it is meaningful, the alignment of a BITFIELD must be one bit; in other words successive BITFIELDs are butted together with no padding bits[1]. Within the limits of what one can extract from BITFIELDs, namely INTEGER VARIETYs, this is how one should implement non-standard alignments, perhaps in constructing data, such as protocols, for exchange between machines. One could implement some Ada representational statements in this way; certainly the most commonly used ones.

1. Note that is not generally true for C bitfields; most C ABIs have (different) rules for putting in padding bits depending on the size of the bitfield and its relation with the natural alignments. This is a fruitful source of errors in data exchange between different C ABIs.

5 Procedures and Locals

All procedures in TDF are essentially global; the only values which are accessible from the body of a procedure are those which are derived from global TAGs (introduced by TAGDEFs or TAGDECs), local TAGs defined within the procedure and parameter TAGs of the procedure

5.1 Defining and calling procedures

5.1.1 make_proc

All executable code in TDF will arise from an EXP PROC made by the **make_proc** (S5.14.50) constructor:

result_shape: SHAPE
params_intro: LIST(TAGSHACC)
var_intro: OPTION(TAGACC)
body: EXP BOTTOM
→ EXP PROC

The *params_intro* and *var_intro* parameters introduce the formal parameters of the procedure which may be used in *body*. The procedure result will have SHAPE *result_shape* and will be given by some **return** construction within *body* (section 4.1.2 on page 19). The basic model is that space will be provided to copy actual parameters (supplied by some **apply_proc** (S5.14.5)) by value into these formals and the body will treat this space as local variables.

The straightforward formal parameters are introduced by a LIST of auxiliary SORT TAGSHACC (S5.36) which gives the TAG to be used for the formal parameter within body together with SHAPE and access information. A TAGSHACC is constructed using make_tagshacc (S5.36.1):

sha: SHAPE
opt_access: OPTION(LIST(ACCESS))
tg_intro: TAG POINTER(alignment(*sha*))
→ TAGSHACC

Within *body*, the formal will be accessed using *tg_intro*; it is always considered to be a pointer to the space of SHAPE *sha* allocated by **apply_proc**, hence the pointer SHAPE.

For example, if we had a simple procedure with one integer parameter, *var_intro* would be empty and *params_intro* might be:

params_intro = **make_tagshacc**(**integer**(v), empty, **make_tag**(13))

Then, TAG 13 from the enclosing UNIT's name-space is identified with the formal parameter with SHAPE POINTER(INTEGER(v)). Any use of **obtain_tag**(**make_tag**(13)) in *body* will deliver a pointer to the integer parameter. I shall return to the meaning of *opt_access* and the ramifications of the scope and extent of TAGs involved in conjunction with local declarations in section 4.2.1 on page 19.

5.1.2 apply_proc, return

Procedures are called[1] using **apply_proc** (S5.14.5):

result_shape: SHAPE
arg1: EXP PROC
arg2: LIST(EXP)
varparam: OPTION(EXP)
$\longrightarrow$ EXP *result_shape*

Here *arg1* is the procedure to be called and *arg2* gives the actual parameters. The SHAPEs of these actuals must correspond exactly to the *params_intro* of the **make_proc** of the procedure. The SHAPE of the result of the call is given by *result_shape* although the *body* of a **make_proc** always has SHAPE BOTTOM; this is because its evaluation normally terminates by giving the result of the procedure using a **return** (S5.14.77):

arg1: EXP *x*
$\rightarrow$ EXP BOTTOM

Here *x* must be identical to the *result_shape* of the call of the procedure There may be several **return**s in body; and the SHAPE *x* in each will be the same as the SHAPE specified in the **apply_proc** and the the **make_proc**.. Some languages allow different types to be returned depending on the particular call. The producer must resolve this issue. For example, C allows one to deliver void if the resulting value is not used. In TDF a dummy value must be prvided at the return; for example **make_value**(*result_shape*).

5.1.3 *vartag*, *varparam*

Use of the *var_option* OPTION in **make_proc** and the corresponding *varparam* in **apply_proc** allows one to have an indefinite number of extra parameters of possibly different SHAPEs to the procedure, where the actual number and SHAPEs can be deduced in some way by the *body* of the **make_proc** (cf. printf in C). One supplies an extra actual parameter, *varparam*, which usually would be a structure grouping the extra parameters. The body of the procedure can then access these values using the pointer given by the TAG given in *var_intro*, using **add_to_ptr** with some computed offsets to pick out the individual fields which are the extra parameters.

5.2 Defining and using locals

5.2.1 identify, variable

Local definitions within the *body* of a procedure are given by two EXP constructors which permit one to give names to values over a scope given by the definition. Note that this is somewhat different to declarations in standard languages where the declaration is usually embedded in a larger construct which defines the scope of the name; here the scope is explicit in the definition. The reason for this will become more obvious in the discussion of TDF transformations. The simpler constructor is **identify** (S5.14.33):

opt_access: OPTION(ACCESS)
name_intro: TAG *x*
definition: EXP *x*
body: EXP *y*
$\rightarrow$ EXP *y*

The *definition* is evaluated and its result is identified with the TAG given by *name_intro* within its scope *body*. Hence the use of any **obtain_tag**(*name_intro*) within *body* is equivalent to using this result. Anywhere else, **obtain_tag**(*name_intro*) is meaningless, including in other procedures.

1. There may be other methods of calling and defining procedures in further extensions to TDF

The other kind of local definition is **variable** (S5.14.84):

opt_access: OPTION(ACCESS)
name_intro: TAG *x*
init: EXP *x*
body: EXP *y*
→ EXP *y*

Here the *init* EXP is evaluated and its result serves as an initialisation of space of SHAPE *x* local to the procedure. The TAG name_intro is then identified with a pointer to that SPACE within body. A use of **obtain_tag**(*name_intro*) within *body* is equivalent to using this pointer and is meaningless outside *body* or in other procedures. Many variable declarations in programs are uninitialised; in this case, the *init* argument could be provided by **make_value** (S5.14.52)which will produce some value with SHAPE given by its parameter.

5.2.2 Locals model and ACCESS

The ACCESS SORT (S5.1) given in tag declarations is a way of describing a list of properties to be associated with the tag. At the moment there are just three possibilities, but others are invisaged in the future[1]. They are **standard_access** (the default) , **visible** and **long_jump_access**. Any of these can be combined using **add_access**.

The basic model used for the locals and parameters of a procedure is a frame within a stack of nested procedure calls. One could implement a procedure by allocating space according to SHAPEs of all of the parameter and local TAGs so that the corresponding values are at fixed offsets from the start of the frame.

Indeed, if the ACCESS *opt_access* parameter in a TAG definition is produced by **visible** (S5.1.6), then a translator is almost bound to do just that for that TAG. This is because it allows for the possibility of the value to be accessed in some way other than by using **obtain_tag** (S5.14.59) which is the standard way of recovering the value bound to the TAG. The principal way that this could happen within TDF is by the combined use of **env_offset** (S5.14.21) to give the offset and **current_env** (S5.14.18) to give a pointer to the current frame. These are effectively defined by the following identities:

If TAG t is derived from a **variable** definition or is a parameter:

add_to_ptr(**current_env**(), **env_offset**(t)) = **obtain_tag**(t)

and if TAG t is derived from an **identify** definition:

contents(shape(t), **add_to_ptr**(**current_env**(), **env_offset**(t))) = **obtain_tag**(t)

These identities are valid throughout the extent of t, including in inner procedure calls. In other words, one can dynamically create a pointer to the value by composing **current_env** and **env_offset**.

The importance of this is that **env_offset**(t) is a constant OFFSET and can be used anywhere with the enclosing CAPSULE[2], in other procedures or as part of constant TAGDEF; remember that the TDFINT underlying t is unique within the UNIT. The result of a **current_env** could be passed to another procedure (as a parameter, say) and this new procedure could then access a local of the original by using its **env_offset**. This would be the method one would use to access non-local, non-global identifiers in a language which allowed one to define procedures within procedures such as Pascal or Algol. Of course, given the stack-based model, the value given by **current_env** becomes meaningless once the procedure in which it is invoked is exited.

The **long_jump_access** flag is used to indicate that the tag must be available after a **long_jump** (S5.14.40)). In practice, if either **visible** or **long_jump_access** is set, most translators would allocate the space for the declaration on the main-store stack rather than in an available register. If it is not set, then a translator is free to use its own criteria for whether space which can fit into a register is allocated on the stack or in a register, provided there is no observable difference (other than time or program size) between the two possibilities.

Some of these criteria are rather obvious; for example, if a pointer to local variable is passed outside the procedure in an opaque manner, then it is highly unlikely that one can allocate the variable in a register. Some might be less obvious. If the only uses of a TAG t was in **obtain_tag**(t)s which are operands of **content**s (S5.14.16) or the

1. For example, “this is a read only pointer”, “this tag is not aliased” etc. would be useful in translator optimisations.
2. If it was not in the current UNIT, one would have to arrange to link it using a LINK.

left-hand operands of **assign**s (S5.14.6), most ABIs would allow the tag to be placed in a register. We do not necessarily have to generate a pointer value if it can be subsumed by the operations available.

5.2.3 local_alloc, local_free_all, last_local

The size of stack frame produced by **variable** and **identify** definitions is a translate-time constant since the frame is composed of values whose SHAPEs are known. TDF also allows one to produce dynamically sized local objects which are conceptually part of the frame. These are produced by **local_alloc** (S5.14.37):

arg1: EXP OFFSET(*x, y*)
→ EXP POINTER(alloca_alignment)

The operand *arg1* gives the size of the new object required and the result is a pointer to the space for this object "on top of the stack" as part of the frame. The quotation marks indicate that a translator writer might prefer to maintain a dynamic stack as well as static one. There are some disadvantages in putting everything into one stack which may well out-weigh the trouble of maintaining another stack which is relatively infrequently used. If a frame has a known size, then all addressing of locals can be done using a stack-front register; if it is dynamically sized, then another frame-pointer register must be used - some ABIs make this easy but not all. The majority of procedures contain no **local_alloc**s, so their addressing of locals can always be done relative to a stack-front; only the others have to use another register for a frame pointer.

The alignment of pointer result is alloca_alignment which must include all SHAPE alignments.

There are two constructors for releasing space generated by **local_alloc**. To release all such space generated in the current procedure one does **local_free_all**() (S5.14.39); this reduces the size of the current frame to its static size.

The other constructor is **local_free** (S5.14.38) whch is effectively a "pop" to **local_alloc**'s "push":

a: EXP OFFSET(*x*, *y*)
p: EXP POINTER(alloca_alignment)
→ EXP TOP

Here *p* must evaluate to a pointer generated either by **local_alloc** or **last_local** (S5.14.36). The effect is to free all of the space locally allocated after p. The usual implementation (with a downward growing stack) of this is that p becomes the "top of stack" pointer[1].

5.3 Heap storage

At the moment, there are no explicit constructors of creating dynamic off-stack storage in TDF. Any off-stack storage requirements must be met by the API in which the system is embedded, using the standard procedural interface. For example, the ANSI C API allows the creation of heap space using standard library procedures like malloc.

1. The specification of TDF will probably be extended to allow more complex means of creating and freeing local dynamic space, so watch this space.

6 Control Flow within procedures

6.1 Unconditional flow

6.1.1 sequence

To perform a sequential set of operations in TDF, one uses the constructor **sequence**(S5.14.79):

statements: LIST(EXP)
result: EXP *x*
→ EXP *x*

Each of the *statements* are evaluated in order, throwing away their results. Then, *result* is evaluated and its result is the result of the **sequence**.

A translator is free to rearrange the order of evaluation if there is no observable difference other than in time or space. This applies anywhere I say "something is evaluated and then ...". We find this kind of statement in definitions of local variables in section 4.2 on page 19, and in the controlling parts of the conditional constructions below.

For a more precise discussion of allowable reorderings see (S7.14) .

6.2 Conditional flow

6.2.1 labelled, make_label

All simple changes of flow of control within a TDF procedure are done by jumps or branches to LABELs, mirroring what actually happens in most computers. There are three constructors which introduce LABELs; the most general is **labelled** (S5.14.35) which allows arbitrary jumping between its component EXPs:

placelabs_intro: LIST(LABEL)
starter: EXP *x*
places: LIST(EXP)
→ EXP *w*

Each of the EXPs in *place*s is labelled by the corresponding LABEL in *place*labs_intro; these LABELs are constructed by **make_label** (S5.19.2) applied to a TDFINT uniquely drawn from the LABEL name-space introduced by the enclosing PROPS. The evaluation starts by evaluating *starter*; if this runs to completion the result of the **labelled** is the result of *starter*. If there is some jump to a LABEL in *placelabs_intro* then control passes to the corresponding EXP in *place*s and so on. If any of these EXPS runs to completion then its result is the result of the **labelled**; hence the SHAPE of the result, w, is the LUB of the SHAPEs of the component EXPs.

Note that control does not automatically pass from one EXP to the next; if this is required the appropriate EXP must end with an explicit **goto**.

6.2.2 goto, make_local_lv, goto_local_lv, long_jump

The unconditional **goto** (S5.14.31)is the simplest method of jumping. In common with all the methods of jumping using LABELs, its LABEL parameter must have been introduced in an enclosing construction, like **labelled**, which scopes it.

One can also pick up a label value of SHAPE POINTER {code} (usually implemented as a program address) using **make_local_lv** (S5.14.44) for later use by an "indirect jump" such as **goto_local_lv** (S5.14.32). Here the same prohibition holds - the construction which introduced the LABEL must still be active.

The construction **goto_local_lv** only permits one to jump within the current procedure; if one wished to do a jump out of a procedure into a calling one, one uses **long_jump** (S5.14.40) which requires a pointer to the destination frame (produced by **current_env** in the destination procedure) as well as the label value. If a **long_jump** is made to a label, only those local TAGs which have been defined with a **visible** ACCESS are guaranteed to have preserved their values; the translator could allocate the other TAGs in scope as registers whose values are not necessarily preserved.

6.2.3 integer_test, NTEST

Conditional branching is provided by the various **_test** constructors, one for each primitive SHAPE except BITFIELD. I shall use **integer_test** (S5.14.34) as the paradigm for them all:

nt: NTEST
dest: LABEL
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
→ EXP TOP

The NTEST *nt* (S5.24) chooses a dyadic test (e.g. =, >=, <, etc.) that is to be applied to the results of evaluating *arg1* and *arg2*. If *arg1 nt arg2* then the result is TOP; otherwise control passes to the LABEL *dest*. In other words, **integer_test** acts like an assertion where if the assertion is false, control passes to the LABEL instead of continuing in the normal fashion.

Some of the constructors for NTESTs are disallowed for some **_test**s (e.g. **proc_test** (S5.14.73)) while others are redundant for some **_tests**; for example, **not_greater_than** (S5.24.9)is the same as **less_than_or_equal** (S5.24.7) for all except possibly **floating_test**. where the use of NaNs (in the IEEE sense) as operands may give different results.

6.2.4 case

There are only two other ways of changing flow of control using LABELs. One arises in ERROR_TREATMENTs which will be dealt with in the arithmetic operations. The other is **case** (S5.14.8):

exhaustive: BOOL
control: EXP INTEGER(*v*)
branches: LIST(CASELIM)
→ EXP (*exhaustive* ? BOTTOM : TOP)

Each CASELIM is constructed using **make_caselim** (S5.12.1):

branch: LABEL
lower: SIGNED_NAT
upper: SIGNED_NAT
→ CASELIM

In the **case** construction, the *control* EXP is evaluated and tested to see whether its value lies inclusively between some *lower* and *upper* in the list of CASELIMs. If so, control passes to the corresponding *branch*. The order in which these tests are performed is undefined, so it is probably best if the tests are exclusive. The exhaustive flag being true asserts that one of the branches will be taken and so the SHAPE of the result is BOTTOM. Otherwise, if none of the branches are taken, its SHAPE is TOP and execution carries on normally.

6.2.5 conditional, repeat

Besides **labelled**, two other constructors, **conditional** and **repeat**, introduce LABELs which can be used with the various jump instructions. Both can be expressed as **labelled**, but have extra constraints which make assertions about the use of the LABELs introduced and could usually be translated more efficiently; hence producers are advised to use them where possible. A conditional expression or statement would usually be done using **conditional** (S5.14.15):

alt_label_intro:	LABEL
first:	EXP *x*
alt:	EXP *y*
	→ EXP(LUB(*x, y*))

Here *first* is evaluated; if it terminates normally, its result is the result of the **conditional**. If a jump to *alt_label_intro* occurs then *alt* is evaluated and its result is the result of the **conditional**. Clearly, this, so far, is just the same as **labelled**((*alt_label_intro*), *first*, (*alt*)). However, **conditional** imposes the constraint that *alt* cannot use *alt_label_intro*. All jumps to *alt_label_intro* are "forward jumps" - a useful property to know in a translator.

Obviously, this kind of conditional is rather different to those found in traditional high-level languages which usually have three components, a boolean expression, a then-part and an else-part. Here, the *first* component includes both the boolean expression and the then-part; usually we find that it is a **sequence** of the tests (branching to *alt_label_intro*) forming the boolean expression followed by the else-part. This formulation means that HLL constructions like "andif" and "orelse" do not require special constructions in TDF.

A simple loop can be expressed using **repeat** (S5.14.76):

repeat_label_intro:	LABEL
start:	EXP TOP
body:	EXP *y*
	→ EXP *y*

The EXP *start* is evaluated, followed by *body* which is labelled by *repeat_label_intro*. If a jump to *repeat_label_intro* occurs in *body*, then *body* is re-evaluated. If *body* terminates normally then its result is the result of the **repeat**. This is just the same as:

labelled((*repeat_label_intro*), **sequence**((*start*), **goto**(*repeat_label_intro*)), (*body*))

except that no jumps to *repeat_label_intro* are allowed in *start* - a useful place to do initialisations for the loop.

Just as with conditionals, the tests for the continuating or breaking the loop are included in *body* and require no special constructions.

7 Values, variables and assignments.

TAGs in TDF fulfil the role played by identifiers in most programming languages. One can apply **obtain_tag** to find the value bound to the TAG. This value is always a constant over the scope of a particular definition of the TAG. This may sound rather strange to those used to the concepts of left-hand and right-hand values in C, for example, but is quite easily explained as follows.

If a TAG, id, is introduced by an **identify**, then the value bound is fixed by its *definition* argument. If, on the other hand, v was a TAG introduced by a **variable** definition, then the value bound to v is a pointer to fixed space in the procedure frame (i.e. the left-hand value in C).

7.1 contents

In order to get the contents of this space (the right-hand value in C), one must apply the **contents** operator to the pointer:

contents(shape(v), **obtain_tag**(v))

In general, the **contents**(S5.14.16) constructor takes a SHAPE and an expression delivering pointer:

s: SHAPE
arg1: EXP POINTER(*x*)
→ EXP *s*

It delivers the value of SHAPE *s*, pointed at by the evaluation of *arg1*. The alignment of *s* need not be identical to *x*. It only needs to be included in it; this would allow one, for example, to pick out the first field of a structure from a pointer to it.

7.2 assign

A simple assignment in TDF is done using **assign** (S5.14.6):

arg1: EXP POINTER(*x*)
arg2: EXP *y*
→ EXP TOP

The EXPs *arg1* and *arg2* are evaluated (no ordering implied) and the value of SHAPE *y* given by *arg2* is put into the space pointed at by *arg1*. Once again, the alignment of *y* need only be included in *x*, allowing the assignment to the first field of a structure using a pointer to the structure. An assignment has no obvious result so its SHAPE is TOP.

Some languages give results to assignments. For example, C defines the result of an assignment to be its right-hand expression, so that if the result of (v = exp) was required, it would probably be expressed as:

identify(empty, newtag, exp,
sequence((**assign**(**obtain_tag**(v), **obtain_tag**(newtag))), **obtain_tag**(newtag)))

From the definition of **assign**, the destination argument, *arg1*, must have a POINTER shape. This means that given the TAG id above, **assign**(**obtain_tag**(id), lhs) is only legal if the *definition* of its **identify** had a POINTER SHAPE. A trivial example would be if id was defined:

identify(empty, id, **obtain_tag**(v), **assign**(**obtain_tag**(id), lhs))

This identifies id with the variable v which has a POINTER SHAPE, and assigns lhs to this pointer. Given that id does not occur in lhs, this is identical to:

assign(**obtain_tag**(v), lhs).

Equivalences like this are widely used for transforming TDF in translators and other tools (see section 10 on page 37).

7.2.1 move_some, TRANSFER_MODE

The value assigned in **assign** has some fixed SHAPE; its size is known at translate-time. Variable sized objects can be moved by **move_some** (S5.14.54):

md: TRANSFER_MODE
arg1: EXP POINTER *x*
arg2: EXP POINTER y
arg3: EXP OFFSET(z, t)
→ EXP TOP

The EXP *arg1* is the destination pointer, and *arg2* is a source pointer. The amount moved is given by the OFFSET *arg3*.

The TRANSFER_MODE *md* parameter controls the way that the move will be performed. With **move_some**, its only significant constructor is **overlap**(S5.48.4). If **overlap** is present, then the translator will ensure that the move is equivalent to moving the source into new space and then copying it to the destination; it would probably do this by choosing a good direction in which to step through the value. The alternative, **standard_transfer_mode**, indicates that it does not matter.

7.3 contents_with_mode, assign_with_mode

There are variants of both the **contents** and **assign** constructors. The signature of **contents_with_mode** is:

md: TRANSFER_MODE
s: SHAPE
arg1: EXP POINTER(*x*)
→ EXP s

Here, the only significant TRANSFER_MODE constructor *md* is **volatile**(S5.48.6). This is principally intended to implement the C volatile construction; it certainly means that the **contents_with_mode** operation will never be "optimised" away.

Similar considerations apply to **assign_with_mode**; here the **overlap** TRANSFER_MODE is also possible with the same meaning as in **move_some**(section 6.2.1 on page 26).

8 Operations

Most of the arithmetic operations of TDF have familiar analogues in standard languages and processors. They differ principally in how error conditions (e.g. numeric overflow) are handled. There is a wide diversity in error handling in both languages and processors, so TDF tries to reduce it to the simplest primitive level compatible with their desired operation in languages and their implementation on processors. Before delving into the details of error handling, it is worthwhile revisiting the SHAPEs and ranges in arithmetic VARIETYs.

8.1 VARIETY and overflow

An INTEGER VARIETY, for example, is defined by some range of signed natural numbers. A translator will fit this range into some possibly larger range which is convenient for the processor in question. For example, the integers with **variety**(1,10) would probably be represented as unsigned characters with range (0..255), a convenient representation for both storage and arithmetic.

The question then arises of what is meant by overflow in an operation which is meant to deliver an integer of this VARIETY - is it when the integer result is outside the range (1..10) or outside the range (0..255)? For purely pragmatic reasons, TDF chooses the latter - the result is overflowed when it is outside its representational range (0..255). If the program insists that it must be within (1..10), then it can always test for it. If the program uses the error handling mechanism and the result is outside (1..10) but still within the representational limits, then, in order for the program to be portable, then the error handling actions must in some sense be "continuous" with the normal action. This would not be the case if, for example, the value was used to index an array with bounds (1..10), but will usually be the case where the value is used in further arithmetic operations which have similar error handling. The arithmetic will continue to give the mathematically correct result provided the representational bounds are not exceeded.

The limits in a VARIETY are there to provide a guide to its representation, and not to give hard limits to its possible values. This choice is consistent with the general TDF philosophy of how exceptions are to be treated. If, for example, one wishes to do array-bound checking, then it must be done by explicit tests on the indices and jumping to some exception action if they fail. Similarly, explicit tests can be made on an integer value, provided its representational limits are not exceeded. It is unlikely that a translator could produce any more efficient code, in general, if the tests were implicit. The representational limits can be exceeded in arithmetic operations, so facilities are provided to allow one to jump to a label if it happens in an operation.

8.1.1 ERROR_TREATMENT

Taking integer addition as an example, **plus** (S5.14.71) has signature:

ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
→ EXP INTEGER(*v*)

The result of the addition has the same integer VARIETY as its parameters. If the representational bounds of *v* are exceeded, then the action taken depends on the ERROR_TREATMENT(S5.13) *ov_err*. The constructor for *ov_err* that one would use if one wished to treat an overflow as an exception is **error_jump** (S5.13.3):

lab: LABEL
→ ERROR_TREATMENT

A branch to *lab* will occur if the result overflows. The action performed there will depend on the program and language involved. If the language were Ada, it would probably jump to some exception-handler, either directly if the error-handler was in the current procedure or indirectly using **long_jump** on some globals if not.

8.1.2 impossible, ignore

There are two other ways of constructing ERROR_TREATMENTs[1]. The constructor, **impossible**, is an assertion by the producer that overflow will not occur; on its head be it if it does. The other is **wrap**; it is only defined for integer varieties. For all the standard integer operations which have ERROR_TREATMENTs, the result is undefined if the lower bound of the VARIETY *v* is negative. If it is defined, then the calculation is done modulo 2 raised to the power of the number of bits of the representational variety; thus, integer arithmetic with byte representational variety is done modulo 256. This just corresponds to what happens in most processors and, incidentally, the definition of C.

8.2 Division and remainder

The various constructors in involving integer division (e.g. **div1** (S5.14.19), **rem1** (S5.14.74)) have two ERROR_TREATMENT parameters, one for overflow and one for divide-by-zero e.g. **div1** is:

div_by_zero_error: ERROR_TREATMENT
ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
→ EXP INTEGER(*v*)

. There are two different kinds of division operators (with corresponding remainder operators) defined. The operators **div2** (S5.14.20)and **rem2** (S5.14.75) are those generally implemented directly by processor instructions giving the sign of the remainder the same as the sign of the quotient. The other pair, **div1** and **rem1**, is less commonly implemented in hardware, but have rather more consistent mathematical properties; here the sign of remainder is the same as the sign of divisor. Thus, **div1**(x, 2) is the same as **shift_right**(x, 1) which is only true for **div2** if x is positive. The two pairs of operations give the same results if both operands have the same sign. The precise definition of the divide operations is given in (S7.4).

1. There may be extensions to this to allow easier use of system provided error traps.

8.3 change_variety

Conversions between the various INTEGER varieties are provided for by **change_variety** (S5.14.12):

ov_err: ERROR_TREATMENT
r: VARIETY
arg1: EXP INTEGER(*v*)
→ EXP INTEGER(*r*)

If the value *arg1* is outside the limits of the representational variety of *r*, then the ERROR_TREATMENT *ov_err* will be invoked.

8.4 and, or, not, xor

The standard logical operations, **and** (S5.14.4), **not** (S5.14.58), **or**(S5.14.70) and **xor** (S5.14.85)are provided for all integer varieties. A purist might argue that these cannot be architecturally independent for signed varieties - what about ones-complement arithmetic? A pedantic producer should indeed only use unsigned varieties with logical operations, but ones-complement machines are few and far between. I wonder how many C programmers know that (x &= -4) does not necessarily remove the bottom two bits from x; it really should have been written (x &= ~3).

8.5 Floating-point operations, ROUNDING_MODE

All of the floating-point operations include ERROR-TREATMENTs. If the result of a floating-point operation cannot be represented in the desired FLOATING_VARIETY, the error treatment is invoked. If the ERROR_TREATMENT is **wrap** or **impossible**, the result is undefined; otherwise the jump operates in the same way as for integer operations. Both **floating_plus**(S5.14.29) and **floating_mult** (S5.14.27) are defined as n-ary operations. In general, floating addition and multiplication are not associative, but a producer may not care about the order in which they are to be performed. Making them appear as though they were associative allows the translator to choose an order which is convenient to the hardware.

Conversions from integer to floating are done by **float_int** (S5.14.23) and from floating to integers by **round_with_mode** (S5.14.78). This latter constructor has a parameter of SORT ROUNDING_MODE (S5.26)which effectively gives the IEEE rounding mode to be applied to the float to produce its integer result.

8.6 change_bitfield_to_int, change_int_to_bitfield

There are two bit-field operation, **change_bitfield_to_int** (S5.14.9)and **change_int_to_bitfield** (S5.14.11) to transform between bit-fields and integers. If the varieties do not fit the result is undefined; the producer can always get it right. The main difficulty for translators dealing with bit-fields is not in these operations themselves, but in the representation and manipulation of pointers to bitfields, i.e. those with SHAPE POINTER(bv) where bv is a BITFIELD_VARIETY. Usually, processors do not have pointers to individual bits, so the representation of a POINTER(bv) must be two words, one giving a byte-address and the other a bit displacement. The choice of the exact details of this can be quite tricky as is the OFFSET arithmetic involved. Current translators go to considerable lengths to avoid ever having to explicitly create one of these compound pointers; they can always be avoided in C since it has no type corresponding to a pointer to a bit-field.

8.7 make_compound, make_nof, n_copies

There is one operation to make values of COMPOUND SHAPE, **make_compound** (S5.14.41):

arg1: EXP OFFSET(*base, y*)
arg2: LIST(EXP)

$\rightarrow$ EXP COMPOUND(*sz*)

The OFFSET *arg1* is evaluated as a translate-time constant to give *sz*, the size of the compound object. The EXPs of arg2 are alternately OFFSETs (also translate-time constants) and values which will be placed at those offsets. This constructor is used to construct values given by structure displays; in C, these only occur with constant *val[i]* in global definitions. It is also used to provide union injectors; here *sz* would be the size of the union and the list would probably two elements with the first being an **offset_zero**.

Constant sized array values may be constructed using **make_nof** (S5.14.45), **make_nof_int** (see section 8.3 on page 32), and **n_copies** (S5.14.56). Again, they only occur in C as constants in global definitions.

9 Constants

The representation of constants clearly has peculiar difficulties in any architecture neutral format. Leaving aside any problems of how numbers are to be represented, we also have the situation where a “constant” can have different values on different platforms. An obvious example would be the size of a structure which, although it is a constant of any particular run of a program, may have different values on different machines. Further, this constant is in general the result of some computation involving the sizes of its components which are not known until the platform is chosen. In TDF, sizes are always derived from some EXP OFFSET constructed using the various OFFSET arithmetic operations on primitives like **shape_offset** and **offset_zero**. Most such EXP OFFSETs produced are in fact constants of the platform; they include field displacements of structure as well as their sizes. TDF assumes that, if these EXPs can be evaluated at translate-time (i.e. when the sizes and alignments of primitive objects are known), then they must be evaluated there. An example of why this is so arises in **make_compound**; the SHAPE of its result EXP depends on its *arg1* EXP OFFSET parameter and all SHAPEs must be translate-time values.

An initialisation of a TAGDEF is a constant in this sense; this allows one to ignore any difficulties about their order of evaluation in the UNIT and consequently the order of evaluation of UNITs. Once again all the EXPs which are initialisations must be evaluated before the program is run. The limitation on an initialisation EXP to ensure this is basically that one cannot take the contents of a variable declared outside the EXP after all tokens and conditional evaluation is taken into account. In other words, each TDF translator effectively has an TDF interpreter which can do evaluation of expressions (including conditionals etc.) involving only constants such as numbers, sizes and addresses of globals. This corresponds very roughly to the kind of initialisations of globals that are permissible in C; for a more precise definition, see (S7.3).

9.1 exp_cond

Another place where translate-time evaluation of constants is mandated is in the various **_cond** constructors which give a kind of “conditional compilation” facility; every SORT which has a SORTNAME, other that TAG, TOKEN and LABEL, has one of these constructors e.g. **exp_cond** (S5.14.2):

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

The constant, *control*, is evaluated at translate time. If it is not zero the entire construction is replaced by the EXP in *e1*; otherwise it is replaced by the one in *e2*. In either case, the other BITSTREAM is totally ignored; it even does not need to be sensible TDF. This kind of construction is use extensively in C pre-processing directives e.g.:

```
#if (sizeof(int) == sizeof(long)) ...
```

9.2 make_int

Integer constants are constructed using **make_int** (S5.14.43):

v: VARIETY
value: SIGNED_NAT
→ EXP INTEGER(*v*)

The SIGNED_NAT *value* is an encoding of the binary value required for the integer; this value must lie within the limits given by *v*. I have been rather slip-shod in writing down examples of integer constants earlier in this document; where I have written 1 as an integer EXP, for example, I should have written **make_int**(v, 1) where v is some appropriate VARIETY.

9.3 TDFSTRING, make_nof_int

Constants for both floats and strings use TDFSTRINGs; a TDFSTRING(*k, n*) is an encoding of *n* unsigned integers of size *k* bits. A constant string is just an particular example of **make_nof_int** (S5.14.46):

v: VARIETY
str: TDFSTRING(*k, n*)
→ EXP NOF(*n*, INTEGER(*v*))

Each unsigned integer in *str* must lie in the variety *v* and the result is the constant array whose elements are the integers considered to be of VARIETY *v*. An ASCI-C constant string might have *v* = **variety**(-128,127) and *k* = 7; however, **make_nof_int** can be used to make strings of any INTEGER VARIETY; a the elements of a Unicode string would be integers of size 16 bits.

9.4 make_floating

A floating constant uses a TDFSTRING which contains the ASCI characters of a expansion of the number to some base in **make_floating** (S5.14.42):

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

For a normal floating point number, each integer in *mantissa* is either the ASCI '.'-symbol or the ASCI representation of a digit of the representation in the given *base*; i.e. if c is the ASCI symbol, the digit value is c-'0'. The resulting floating point number has SHAPE FLOATING(f) and value $mantissa * base^{exponent}$ rounded according to *rm*. Usually the base will be 10 (sometimes 2) and the rounding mode **to_nearest**. Any floating-point evaluation of expressions done at translate-time will be done to an accuracy greater that implied by the FLOATING_ VARIETY involved, so that floating constants will be as accurate as the platform permits.

9.5 make_null_ptr, make_null_local_lv, make_null_proc

Constants are also provided to give unique null values for pointers, label values and procs i.e.: **make_null_ptr** (S5.14.49), **make_null_local_lv** (S5.14.47)and **make_null_proc** (S5.14.48). Any significant use of these values (e.g. taking the contents of a null pointer) is undefined, but they can be assigned and used in tests in the normal way.

10 Tokens and APIs

All of the examples of the use of TOKENs so far given have really been as abbreviations for commonly used constructs, e.g. the EXP OFFSETS for fields of structures. However, the real justification for TOKENs are their use as abstractions for things defined in libraries or application program interfaces (APIs).

10.1 Application programming interfaces

APIs usually do not give complete language definitions of the operations and values that they contain; generally, they are defined informally in English giving relationships between the entities within them. An API designer should allow implementors the opportunity of choosing actual definitions which fit their hardware and the possibility of changing them as better algorithms or representations become available.

The most commonly quoted example is the representation of the type FILE and its related operations in C. The ANSI C definition gives no common representation for FILE; its implementation is defined to be platform-dependent. A TDF producer can assume nothing about FILE; not even that it is a structure. The only things that can alter or create FILEs are also entities in the Ansi-C API and they will always refer to FILEs via a C pointer. Thus TDF abstracts FILE as a SHAPE TOKEN with no parameters, **make_tok**(T_FILE) say. Any program that uses FILE would have to include a TOKDEC introducing T_FILE:

> **make_tokdec**(T_FILE, **shape**())

and anywhere that it wished to refer to the SHAPE of FILE it would do:

> **shape_apply_token**(**make_tok**(T_FILE), ())

Before this program is translated on a given platform, the actual SHAPE of FILE must be supplied. This would be done by linking a TDF CAPSULE which supplies the TOKDEF for the SHAPE of FILE which is particular to the target platform.

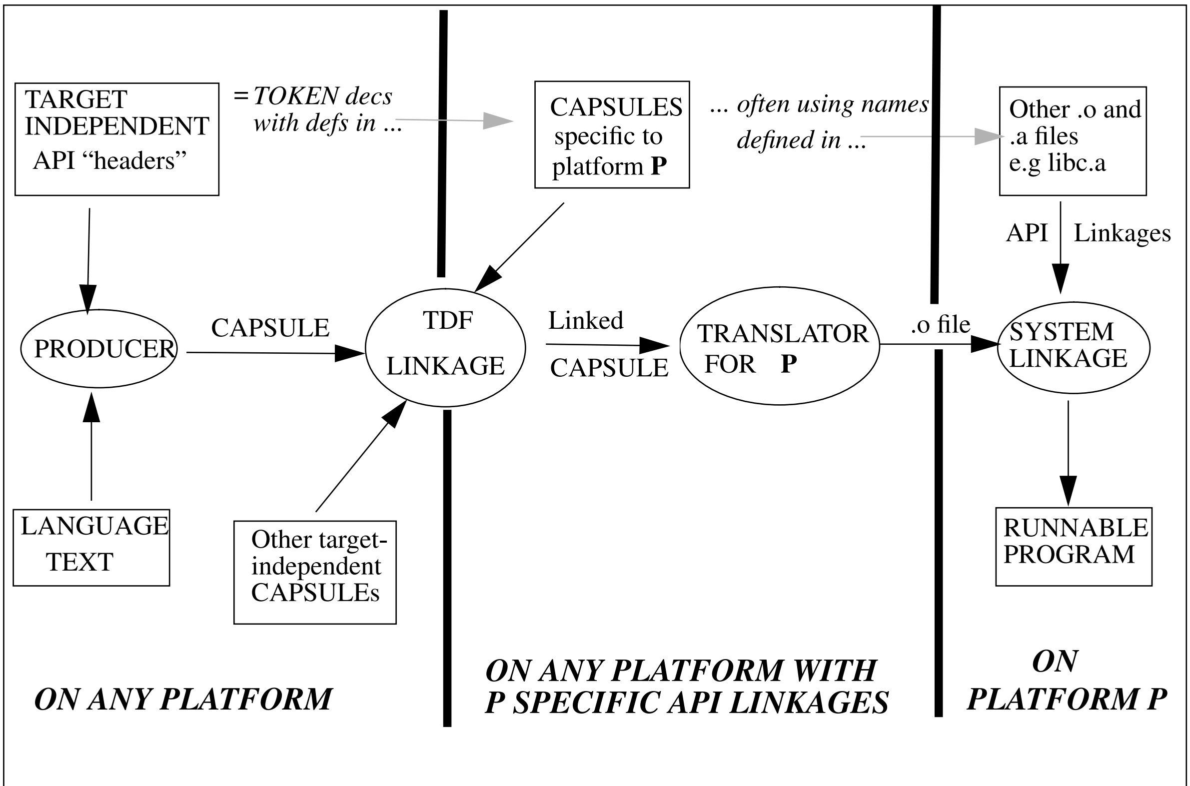
Many of the C operations which use FILEs are explicitly allowed to be expanded as either procedure calls or as macros. For example, putc(c,f) may be implemented either as a procedure call or as the expansion of macro which uses the fields of f directly. Thus, it is quite natural for putc(c, f) to be represented in TDF as an EXP TOKEN with two EXP parameters which allows it to be expanded in either way. Of course, this would be quite distinct from the use of putc as a value (as a proc parameter of a procedure for example) which would require some other representation. One such representation that comes to mind might be to simply to make a TAGDEC for the putc value, supplying its TAGDEF in the Ansi API CAPSULE for the platform. This might prove to be rather short-sighted, since it denies us the possibility that the putc value itself might be expanded from other values and hence it would be better as another parameterless TOKEN. I have not come across an actual API expansion for the putc value as other than a simple TAG; however the FILE* value stdin is sometimes expressed as:

```
#define stdin &_iob[0]
```

which illustrates the point. It is better to have all of the interface of an API expressed as TOKENs to give both generality and flexibility across different platforms.

10.2 Linking to APIs

In general, each API requires platform-dependent definitions to be supplied by a combination of TDF linking and system linking for that platform. This is illustrated in the following diagram giving the various phases involved in producing a runnable program.



TDF PRODUCTION, LINKING AND TRANSLATING

There will be CAPSULEs for each API on each platform giving the expansions for the TOKENs involved, usually as uses of identifiers which will be supplied by system linking from some libraries. These CAPSULEs would be derived from the header files on the platform for the API in question, usually using some automatic tools. For example, there will be a TDF CAPSULE (derived from <stdio.h>) which defines the TOKEN T_FILE as the SHAPE for FILE, together with definitions for the TOKENs for putc, stdin, etc., in terms of identifiers which will be found in the library libc.a.

10.2.1 Target independent headers, unique_extern

Any producer which uses an API will use system independent information to give the common interface TOKENs for this API. In the C producer, this is provided by header files using pragmas, which tell the producer which TOKENs to use for the particular constructs of the API . In any target-independent CAPSULE which uses the API, these TOKENs would be introduced as TOKDECs and made globally accessible by using **make_linkextern** (S5.21.1). For a world-wide standard API, the EXTERNAL(S5.15) “name” for a TOKEN used by **make_linkextern** should be provided by an application of **unique_extern (S5.15.2)** on a UNIQUE drawn from a central repository of names for entities in standard APIs; this repository would form a kind of super-standard for naming conventions in all possible APIs. The mechanism for controlling this super-standard has yet to be set up, so at the moment all EXTERN names are created by **string_extern** (S5.15.1).

An interesting example in the use of TOKENs comes in abstracting field names. Often, an API will say something like "the type Widget is a structure with fields alpha, beta ..." without specifying the order of the fields or whether the list of fields is complete. The field selection operations for Widget should then be expressed using EXP OFFSET TOKENs; each field would have its own TOKEN giving its offset which will be filled in when the target is known. This gives implementors on a particular platform the opportunity to reorder fields or add to them as they like; it also allows for extension of the standard in the same way.

The most common SORTs of TOKENs used for APIs are SHAPEs to represent types, and EXPs to represent values, including procedures and constants. NATs and VARIETYs are also sometimes used where the API does not specify the types of integers involved. The other SORTs are rarely used in APIs; indeed it is difficult to imagine *any* realistic use of TOKENs of SORT BOOL. However, the criterion for choosing which SORTs are available for TOKENisation is not their immediate utility, but that the structural integrity and simplicity of TDF is maintained. It is fairly obvious that having BOOL TOKENs will cause no problems, so we may as well allow them.

10.3 Language programming interfaces

So far, I have been speaking as though a TOKENised API could only be some library interface, built on top of some language, like xpg3, posix, X etc. on top of C. However, it is possible to consider the constructions of the language itself as ideal candidates for TOKENisation. For example, the C for-statement could be expressed as TOKEN with four parameters[1]. This TOKEN could be expanded in TDF in several different ways, all giving the correct semantics of a for-statement. A translator (or other tools) could choose the expansion it wants depending on context and the properties of the parameters. The C producer could give a default expansion which a lazy translator writer could use, but others might use expansions which might be more advantageous. This idea could be extended to virtually all the constructions of the language, giving what is in effect a C-language API; perhaps this might be called more properly a language programming interface (LPI). Thus, we would have TOKENs for C for-statements, C conditionals, C procedure calls, C procedure definitions etc.[2].

The notion of a producer for any language working to an LPI specific to the constructs of the language is very attractive. It could use different TOKENs to reflect the subtle differences between uses of similar constructs in different languages which might be difficult or impossible to detect from their expansions, but which could allow better optimisations in the object code. For example, Fortran procedures are slightly different from C procedures in that they do not allow aliasing between parameters and globals. While application of the standard TDF procedure calls would be semantically correct, knowledge of that the non-aliasing rule applies would allow some procedures to be translated to more efficient code. A translator without knowledge of the semantics implicit in the TOKENs involved would still produce correct code, but one which knew about them could take advantage of that knowledge.

I also think that LPIs would be a very useful tool for crystalising ideas on how languages should be translated, allowing one to experiment with expansions not thought of by the producer writer. This decoupling is also an escape clause allowing the producer writer to defer the implementation of a construct completely to translate-time or link-time, as is done at the moment in C for off-stack allocation. As such it also serves as a useful test-bed for TOKEN constructions which may in future become new constructors of core TDF.

1. Exercise for the reader: what are the SORTs of these parameters?
2. The current C producer does this for some of the constructs, but not in any systematic manner; perhaps it will change.

11 TDF transformations

TDF to TDF transformations form the basis of most of the tools of TDF, including translators. TDF has a rich set of easily performed transformations; this is mainly due to its algebraic nature, the liberality of its sequencing rules, and the ease with which one can introduce new names over limited scopes. For example, a translator is always free to transform:

assign(e1, e2)

to:

identify(empty, new_tag, e1, **assign**(**obtain_tag**(new_tag), e2))

i.e. identify the evaluation of the left-hand side of the assignment with a new TAG and use that TAG as the left-hand operand of a new assignment in the body of the identification. Note that the reverse transformation is only valid if the evaluation of e1 does not side-effect the evaluation of e2. A producer would have had to use the second form if it wished to evaluate e1 before e2. The definition of **assign** allows its operands to be evaluated in any order while **identify** insists that the evaluation of its *definition* is conceptually complete before starting on its *body*.

Why would a translator wish to make the more complicated form from the simpler one? This would usually depend on the particular forms of e1 and e2 as well as the machine idioms available for implementing the assignment. If, for example, the joint evaluation of e1 and e2 used more evaluation registers than is available, the transformation is probably a good idea. It would not necessarily commit one to putting the new tag value into the stack; some other more global criteria might lead one to allocate it into a register disjoint from the evaluation registers. In general, this kind of transformation is used to modify the operands of TDF constructions so that the code-production phase of the translator can just "churn the handle" knowing that the operands are already in the correct form for the machine idioms.

Transformations like this are also used to give optimisations which are largely independent of the target architecture. In general, provided that the sequencing rules are not violated, any EXP construction, F(X), say, where X is some inner EXP, can be replaced by:

identify(empty, new_tag, X, F(**obtain_tag**(new_tag))).

This includes the extraction of expressions which are constant over a loop; if F was some **repeat** construction and one can show that the EXP X is invariant over the repeat, the transformation does the constant extraction.

Most of the transformations performed by translators are of the above form, but there are many others. Particular machine idioms might lead to transformations like changing a test (i>=1) to (i>0) because the test against zero is faster; replacing multiplication by a constant integer by shifts and adds because multiplication is slow; and so on. Target independent transformations include things like procedure inlining and loop unrolling. Often these target independent transformations can be profitably done in terms of the TOKENs of an LPI; loop unrolling is an obvious example.

11.1 Transformations as definitions

As well being a vehicle for expressing optimisation, TDF transformations can be used as the basis for defining TDF. In principle, if we were to define all of the allowable transformations of the TDF constructions, we would have a complete definition of TDF as the initial model of the TDF algebra. This would be a fairly impracticable project, since the totality of the rules including all the simple constructs would be very unwieldy, difficult to check for inconsistencies and would not add much illumination to the definition. However, knowledge of allowable transformations of TDF can often answer questions of the meaning of diverse constructs by relating them to a single construct. What follows is an alphabet of generic transformations which can often help to answer knotty questions. Here, E[X \ Y] denotes an EXP E with all internal occurrences of X replaced by Y.

{A} If F is any non order-specifying[1] EXP constructor and E is one of the EXP operands of F, then:
F(... , E, ...) $\Rightarrow$ **identify**(empty, newtag, E, F(... , **obtain_tag**(newtag), ...))

{B} If E is a non side-effecting[2] EXP and none of the variables used in E are assigned to in B:
identify(v, tag, E, B) $\Rightarrow$ B[**obtain_tag**(tag) \ E]

{C} If all uses of tg in B are of the form **contents**(shape(E), **obtain_tag**(tg)):
variable(v, tg, E, B)
$\Rightarrow$ **identify**(v, nt, E, B[**contents**(shape(E), **obtain_tag**(tg)) \ **obtain_tag**(nt)])

{D} **sequence**((S_1, ... , S_n), **sequence**((P_1, ..., P_m), R)
$\Leftrightarrow$ **sequence**((S_1, ..., S_n, P_1, ..., P_m), R)

{E} If S_i = **sequence**((P_1, ..., P_m), R) :
sequence((S_1, ... , S_n), T)
$\Leftrightarrow$ **sequence**((S_1, ..., S_{i-1}, P_1, ..., P_m, R, S_{i+1}, ..., S_n), T)

{F} E $\Leftrightarrow$ **sequence**(( ), E)

{G} If D is either **identify** or **variable**:
D(v, tag, **sequence**((S_1, ..., S_n), R), B) $\Rightarrow$ **sequence**((S_1, ..., S_n), D(v, tag, R, B))

{H} If S_i is an EXP BOTTOM , then:
sequence((S_1, S_2, ... S_n), R) $\Rightarrow$ **sequence**((S_1, ... S_{i-1}), S_i)

{I} If E is an EXP BOTTOM, and if D is either **identify** or **variable**:
D(v, tag, E, B) $\Rightarrow$ E

{J} If S_i is **make_top**(), then:
sequence((S_1, S_2, ... S_n), R) $\Leftrightarrow$ **sequence**((S_1, ... S_{i-1}, S_{i+1}, ...S_n), R)

{K} If S_n is an EXP TOP:
sequence((S_1, ... S_n), **make_top**()) $\Leftrightarrow$ **sequence**((S_1, ..., S_{n-1}), S_n)

{L} If E is an EXP TOP and E is not side-effecting then
E $\Rightarrow$ **make_top**()

{M} If C is some non order-specifying and non side-effecting constructor, and S_i is C(P_1,..., P_m) where $P_{1..m}$ are the EXP operands of C:
sequence((S_1, ..., S_n), R) $\Rightarrow$**sequence**((S_1, ..., S_{i-1}, P_1, ..., P_m, S_{i+1}, ..., S_n), R)

{N} If none of the S_i use the label L:
conditional(L, **sequence**((S_1, ..., S_n), R), A)
$\Rightarrow$ **sequence**((S_1, ..., S_n), **conditional**(L, R, A))

{O} If there are no uses of L in X[3]:
conditional(L, X, Y) $\Rightarrow$ X

{P} **conditional**(L, E , **goto**(Z)) $\Rightarrow$ E[L \ Z]

{Q} If EXP X contains no use of the LABEL L:
conditional(L, **conditional**(M, X, Y), Z)
$\Rightarrow$ **conditional**(M, X, **conditional**(L, Y, Z))

---

1. The order-specifying constructors are conditional, identify, repeat, labelled, sequence and variable
2. A sufficient condition for not side-effecting in this sense is that there are no apply_procs or local_allocs in E; that any assignments in E are to variables defined in E; and that any branches in E are to labels defined in conditionals in E
3. There are analogous rules for labelled and repeat with unused LABELs.

{R} **repeat**(L, I, E) ⟹ **sequence**((I), **repeat**(L, **make_top**(), E))

{S} **repeat**(L, **make_top**(), E) ⟹ **conditional**(Z, E[L \ Z], **repeat**(L, **make_top**(), E))

{T} If there are no uses of L in E:

repeat(L, **make_top**(), **sequence**((S, E), **make_top**())
⟹ **conditional**(Z, S[L \ Z],
repeat(L, **make_top**(), **sequence**((E, S), **make_top**())))

{U} If f is a procedure defined[1] as:

make_proc(rshape, $formal_{1..n}$, vtg , B(**return** R_1, ..., **return** R_m))
where:
$formal_i$ = **make_tagshacc**(s_i, v_i, tg_i)
and B is an EXP with all of its internal **return** constructors indicated parametrically
then, if A_i has SHAPE s_i
apply_proc(rshape, f, (A_1, ... , A_n), V)
⟹ **variable**(empty, newtag, **make_value**((rshape=BOTTOM)? TOP: rshape),
labelled((L),
variable(v_1, tg_1, A_1, ... , **variable**(v_n, tg_n, A_n,
variable(empty, vtg, V,
B(**sequence**(**assign**(**obtain_tag**(newtag), R_1), **goto**(L)) , ... ,
sequence(**assign**(**obtain_tag**(newtag), R_m), **goto**(L))
)
)
),
contents(rshape, **obtain_tag**(newtag))
)
)

{V} **assign**(E, **make_top**()) ⟹ **sequence**((E), **make_top**())

{W} **contents**(TOP, E) ⟹ **sequence**((E), **make_top**())

{X} **make_value**(TOP) ⟹ **make_top**()

{Y} **component**(s, **contents**(COMPOUND(S), E), D)
⟹ **contents**(s, **add_to_ptr**(E, D))

{Z} **make_compound**(S, ((E_1, D_1), ..., (E_n, D_n)))
⟹ **variable**(empty, nt, **make_value**(COMPOUND(S)),
sequence(
(**assign**(**add_to_ptr**(**obtain_tag**(nt), D_1), E_1),
... ,
assign(**add_to_ptr**(**obtain_tag**(nt), D_n), E_n)),
contents(S, **obtain_tag**(nt))
)
)

11.1.1 Examples of transformations

Any of these transformations may be performed by the TDF translators. The most important is probably {A} which allows one to reduce all of the EXP operands of suitable constructors to **obtain_tag**s. The expansion rules for identification, {G}, {H} and {I}, gives definition to complicated operands as well as strangely formed ones,

1. This has to be modified if B contains any uses of local_free_all or last_local.

e.g. **return**(... **return**(X)...). Rule {A} also illustrates neatly the lack of ordering constraints on the evaluation of operands. For example, **mult**(et, exp1, exp2) could be expanded by applications of {A} to either:

```
identify(empty, t1, exp1,
    identify(empty, t2, exp2, mult(et, obtain_tag(t1), obtain_tag(t2))) )
```

or:

```
identify(empty, t2, exp2,
    identify(empty, t1, exp1, mult(et, obtain_tag(t1), obtain_tag(t2))) )
```

Both orderings of the evaluations of exp1 and exp2 are acceptable, regardless of any side-effects in them. There is no requirement that both expansions should produce the same answer for the multiplications; the only person who can say whether either result is “wrong” is the person who specified the program.

Many of these transformations often only come into play when some previous transformation reveals some otherwise hidden information. For example, after procedure in-lining given by {U} or loop un-rolling given by {S}, a translator can often deduce the behaviour of a **_test** constructor, replacing it by either a **make_top** or a **goto**. This may allow one to apply either {J} or {H} to eliminate dead code in sequences and in turn {N} or {P} to eliminate entire conditions and so on.

Application of transformations can also give expansions which are rather pathological in the sense that a producer is very unlikely to form them. For example, a procedure which returns no result would have result statements of the form **return**(**make_top**()). In-lining such a procedure by {U} would have a form like:

```
variable(empty, nt, make_shape(TOP),
    labelled( (L),
    ... sequence((assign(obtain_tag(nt), make_top())), goto(L)) ...
    contents(TOP, obtain_tag(nt))
    )
)
```

The rules {V}, {W} and {X} allow this to be replaced by:

```
variable(empty, nt, make_top(),
    labelled( (L),
    ... sequence((obtain_tag(nt)), goto(L)) ...
    sequence((obtain_tag(nt)), make_top())
    )
)
```

The **obtain_tag**s can be eliminated by rule {M} and then the **sequence**s by {F}. Sucessive applications of {C} and {B} then give:

```
labelled( (L),
    ... goto(L) ...
    make_top()
    )
```

11.1.2 Programs with undefined values

The definitions of most of the constructors in the TDF specification are predicated by some conditions; if these conditions are not met the effect and result of the constructor is not defined for all possible platforms[1]. Any value which is dependent on the effect or result of an undefined construction is also undefined. This is not the same as

1. However, we may find that the mapping of a constraint allows extra relationships for a class of architectures which do not hold in all generality; this may mean that some constructions are defined on this class while still being undefined in others. (see section 12 on page 44)

saying that a program is undefined if it can construct an undefined value - the dynamics of the program might be such that the offending construction is never obeyed.

12 TDF expansions of offsets

Consider the C structure defined by:

```
typedef struct{ int i; double d; char c;} mystruct;
```

Given that sh_int, sh_char and sh_double are the SHAPEs for int, char and double, the SHAPE of *mystruct* is constructed by:

SH_mystruct = **compound**(S_c)
where:
S_c = **offset_add**(O_c, **shape_offset**(sh_char))
where:
O_c = **offset_pad**(**alignment**(sh_char), S_d)
where:
S_d = **offset_add**(O_d, **shape_offset**(sh_double))
where:
O_d = **offset_pad**(**alignment**(sh_double), S_i)
where[1]:
S_i = **offset_add**(O_i, **shape_offset**(sh_int))
and:
O_i = **offset_zero**(**alignment**(sh_int))

Each of S_c, S_d and S_i gives the minimum "size" of the space required to upto and including the field c, d and i respectively. Each of O_c, O_d and O_i gives the OFFSET "displacement" from a pointer to a *mystruct* required to select the fields c, d and i respectively. The C program fragment:

```
mystruct s;
.... s.d = 1.0; ...
```

would translate to something like:

variable(empty, tag_s, **make_value**(**compound**(S_c)),
  sequence(...
   assign(**add_to_ptr**(**obtain_tag**(tag_s), O_d), 1.0)
   ...
)
)

Each of the OFFSET expressions above are ideal candidates for tokenisation; a producer would probably define tokens for each of them and use **exp_apply_token** to expand them at each of their uses.

From the definition, we find that:

S_c = **shape_offset**(SH_mystruct)
i.e. an OFFSET(**alignment**(sh_int) $\cup$ **alignment**(sh_char) $\cup$ **alignment**(sh_double), {})

1. I could equally have given simply shape_offset(sh_int) for S_i, but the above formulation is more uniform with respect to selection OFFSETs.

This would not be the OFFSET required to describe *sizeof(mystruct)* in C, since this is defined to be the difference between successive elements an array of *mystruct*s. The *sizeof* OFFSET would have to pad S_c to the alignment of SH_mystruct:

offset_pad(**alignment**(SH_mystruct), S_c)

This is the OFFSET that one would use to compute the displacement of an element of an array of *mystruct*s using **offset_mult** with the index.

The most common use of OFFSETs is in **add_to_ptr** to compute the address of a structure or array element. Looking again at its signature in a slightly different form:

arg1: EXP POINTER($y \cup A$)
arg2: EXP OFFSET(y, z)
$\rightarrow$ EXP POINTER(z)
... for any ALIGNMENT A

one sees that *arg2* can measure an OFFSET from a value of a "smaller" alignment than the value pointed at by *arg1*. If *arg2* were O_d, for example, then *arg1* could be a pointer to any structure of the form struct {int i, double d,...} not just *mystruct*. The general principle is that an OFFSET to a field constructed in this manner is independent of any fields after it, corresponding to normal usage in both languages and machines. A producer for a language which conflicts with this would have to produce less obvious TDF, perhaps by re-ordering the fields, padding the offsets by later alignments or taking maxima of the sizes of the fields.

13 Models of the TDF algebra

TDF is a multi-sorted abstract algebra. Any implementation of TDF is a model of this algebra, formed by a mapping of the algebra into a concrete machine. An algebraic mapping gives a concrete representation to each of the SORTs in such a way that the representation of any construction of TDF is independent of context; it is a homomorphism. In other words if we define the mapping of a TDF constructor, C, as MAP[C] and the representation of a SORT, S, as REPR[S] then:

$$\text{REPR}[\ C(P_1 ,..., P_n)\ ] = \text{MAP}[C](\ \text{REPR}(P_1) ,..., \text{REPR}(P_n))$$

Any mapping has to preserve the equivalences of the abstract algebra, such as those exemplified by the transformations {A} - {Z} in section 10.1 on page 37. Similarly, the mappings of any predicates on the constructions, such as those giving “well-formed” conditions, must be satisfied in terms of the mapped representations.

In common with most homomorphisms, the mappings of constructions can exhibit more equivalences than are given by the abstract algebra. The use of these extra equivalences is the basis of most of the target-dependent optimisations in a TDF translator; it can make use of “idioms” of the target architecture to produce equivalent constructions which may work faster than the “obvious” translation. In addition, we may find that may find that more predicates are satisfied in a mapping than would be in the abstract algebra. A particular concrete mapping might allow more constructions to be well-formed than are permitted in the abstract; a producer can use this fact to target its output to a particular class of architectures. In this case, the producer should produce TDF so that any translator not targeted to this class can fail gracefully.

Giving a complete mapping for a particular architecture here is tantamount to writing a complete translator. However, the mappings for the small but important sub-algebra concerned with OFFSETs and ALIGNMENTs illustrates many of the main principles. What follows is two sets of mappings for disparate architectures; the first gives a more or less standard meaning to ALIGNMENTs but the second may be less familiar.

13.1 Model for a 32-bit standard architecture

Almost all current architectures use a “flat-store” model of memory. There is no enforced segregation of one kind of data from another - in general, one can access one unit of memory as a linear offset from any other. Here, TDF ALIGNMENTs are a reflection of constraints for the efficient access of different kinds of data objects - usually one finds that 32-bit integers are most efficiently accessed if they start at 32 bit boundaries and so on.

13.1.1 Alignment model

The representation of ALIGNMENT in a typical standard architecture is a single integer where:

REPR [{ }] = 1
REPR[{bitfield}] = 1
REPR[{char_variety}] = 8
REPR[{short_variety}] = 16

Otherwise, for all other primitive ALIGNMENTS a:

REPR [{a}] = 32

The representation of a compound ALIGNMENT is given by:

$$\text{REPR}[\ A \cup B\ ] = \text{Max}(\text{REPR}[\ A\ ] , \text{REPR}[\ B\ ])$$

i.e. MAP[unite_alignment] = Max

while the ALIGNMENT inclusion predicate is given by:

REPR[A $\supset$ B]= REPR[A] $\geq$ REPR[B }

All the constructions which make ALIGNMENTs are represented here and they will always reduce to an integer known at translate-time. Note that the mappings for $\cup$ and $\supset$ must preserve the basic algebraic properties derived from sets; for example the mapping of $\cup$ must be idempotent, commutative and associative, which is true for Max.

13.1.2 Offset and pointer model

Most standard architectures use byte addressing; to address bits requires more complication. Hence, a value with SHAPE POINTER(A) where REPR(A) $\neq$ 1 is represented by a 32-bit byte address. However, a value with SHAPE POINTER(A) where REPR(A) = 1 (i.e. a pointer to a bitfield) is represented by a pair consisting of a 32-bit byte address and a 32-bit bit-displacement. Denote this pair by:

(a: byte_address, d: bit_displacement)

A value with SHAPE OFFSET(A, B) where REPR(A) $\neq$ 1 is represented by a 32-bit byte-offset.

A value with SHAPE OFFSET(A, B) where REPR(A) = 1 is represented by a 32-bit bit-offset.

13.1.3 Size model

In principle, the representation of a SHAPE is a pair of an ALIGNMENT and a size, given by shape_offset applied to the SHAPE. This pair is constant which can be evaluated at translate time. The construction, shape_ offset(S), has SHAPE OFFSET(alignment(s), { }) and hence is represented by a bit-offset:

REPR[**shape_offset**(**integer**(char_variety))] = 8

REPR[**shape_offset**(**integer**(short_variety))] = 16

.... etc. for other numeric varieties

REPR[**shape_offset**(**ointer**(A))]= (REPR[A] = 1)? 64 : 32

REPR[**shape_offset**(**compound**(E))] = REPR[E]

REPR[**shape_offset**(**bitfield**(**bfvar_bits**(b, N)))] = N

REPR[**shape_offset**(nof(N, S))] = N * REPR[**offset_pad**(**alignment**(S), **shape_offset**(S))]

REPR[**shape_offset**(**top**())] = 0

13.1.4 Offset arithmetic

Using the notation X: S to mean that X is an EXP of SHAPE S, the mappings of the other offset constructors are:

REPR [**offset_zero**(A)] = 0 for all A

REPR[**offset_add**(X: OFFSET(A, C), Y:OFFSET(D, B))] = 8*REPR[X] + REPR[Y]
if REPR[C] $\neq$ 1 $\wedge$ REPR[B] = 1

Otherwise:

REPR[**offset_add**(X, Y)] = REPR[X] + REPR[Y]

REPR[**offset_pad**(A, X: OFFSET(B, C))] = 8*REPR[X]
if REPR[A]= 1 $\wedge$ REPR[C] $\neq$ 1

Otherwise:

REPR[**offset_pad**(A, X: OFFSET(B, C))] = (REPR[X] + REPR[A] -1) / REPR[A]

REPR[**offset_max**(X: OFFSET(A, B), Y: OFFSET(C, D))]
    = Max(REPR[X], 8*REPR[Y]
    if REPR[B] = 1 $\wedge$ REPR[D] $\neq$ 1
REPR[**offset_max**(X: OFFSET(A, B), Y: OFFSET(C, D))]
    = Max(8*REPR[X], REPR[Y]
    if REPR[D] = 1 $\wedge$ REPR[B] $\neq$ 1
Otherwise:
REPR[**offset_max**(X, Y)] = Max(REPR[X], REPR[Y])

The other OFFSET constructors map straightforwardly as in:

REPR[**offset_mult**(X, E)] = REPR[X] * REPR[E]

The mapping of add_to_ptr is given by:

REPR[**add_to_ptr**(E: POINTER A, X: OFFSET(B, C))] = REPR[E] + REPR[X]
        if REPR[A] $\neq$ 1 $\wedge$ REPR[C] $\neq$ 1
REPR[add_to_ptr(E: POINTER A, X: OFFSET(B, C))]
    = (a: REPR[E], d: REPR[X])
    if REPR[A] $\neq$ 1 $\wedge$ REPR[C] = 1
REPR[add_to_ptr(E: POINTER A, X: OFFSET(B, C))]
    = (a: REPR[E].a, d: REPR[E].d + REPR[X])
    if REPR[A] = 1 $\wedge$ REPR[C] = 1

A translator working to this model maps ALIGNMENTs into the integers and their inclusion constraints into numerical comparisons. As a result, it will correctly allow many OFFSETs which are disallowed in general; for example, OFFSET({pointer}, {char_variety}) is allowed since REPR[{pointer}] $\geq$ REPR[{char_variety}]. Rather fewer of these extra relationships are allowed in the next model considered.

13.2 Model for machines like the iAPX-432

The iAPX-432 does not have a linear model of store. The address of a word in store is a pair consisting of a block-address and a displacement within that block. In order to take full advantage of the protection facilities of the machine, block-addresses are strictly segregated from scalar data like integers, floats, displacements etc. There are at least two different kind of blocks, one which can only contain block-addresses and the other which contains only scalar data. There are clearly difficulties here in describing data-structures which contain both pointers and scalar data.

Let us assume that the machine has bit-addressing to avoid the bit complications already covered in the first model. Also assume that instruction blocks are just scalar blocks and that block addresses are aligned on 32-bit boundaries.

13.2.1 Alignment model

An ALIGNMENT is represented by a pair consisting of an integer, giving the natural alignments for scalar data, and boolean to indicate the presence of a block-address. Denote this by:

(s: alignment_of_scalars, b: has_blocks)

We then have:

REPR[**alignment**({ })] = (s: 1, b: FALSE)

REPR[**alignment**({char_variety}) = (s: 8, b:FALSE)

... etc. for other numerical and bitfield varieties.

REPR[**alignment**({pointer})] = (s: 32, b: TRUE)

REPR[**alignment**({proc})] = (s: 32, b: TRUE)

REPR[**alignment**({local_label_value})] = (s: 32, b: TRUE)

The representation of a compound ALIGNMENT is given by:

REPR[A $\cup$ B] = (s: Max(REPR[A].s, REPR[B].s), b: REPR[A].b $\vee$ REPR[B].b)

and their inclusion relationship is given by:

REPR[A $\supset$ B] = (REPR[A].s $\geq$ REPR[B].s) $\wedge$ (REPR[A].b $\vee$ $\neg$ REPR[B].b)

13.2.2 Offset and pointer model

A value with SHAPE POINTER A where $\neg$ REPR[A].b is represented by a pair consisting of a block-address of a scalar block and an integer bit-displacement within that block. Denote this by:

(sb: scalar_block_address, sd: bit_displacement)

A value with SHAPE POINTER A where REPR[A].b is represented by a quad word consisting of two block-addresses and two bit-displacements within these blocks. One of these block addresses will contain the scalar information pointed at by one of the bit-displacements; similarly, the other pair will point at the block addresses in the data are held. Denote this by:

(sb: scalar_block_address, ab: address_block_address,
sd: scalar_displacement, ad: address_displacement)

A value with SHAPE OFFSET(A, B) where $\neg$ REPR[A].b is represented by an integer bit-displacement.

A value with SHAPE OFFSET(A, B) where REPR[A].b is represented by a pair of bit-displacements, one relative to a scalar-block and the other to an address-block. Denote this by:

(sd: scalar_displacement, ad: address_displacement)

13.2.3 Size model

The sizes given by shape_offset are now:

REPR[**shape_offset**(**integer**(char_variety))] = 8

... etc. for other numerical and bitfield varieties.

REPR[**shape_offset**(**pointer**(A))] = (REPR[A].b) ? (sd: 64, ad: 64) : (sd: 32, ad: 32)

REPR[**shape_offset**(**offset**(A, B))] = (REPR[A].b) ? 64 : 32)

REPR[**shape_offset**(**proc**)] = (sd: 32, ad: 32)

REPR[**shape_offset**(**compound**(E))] = REPR[E]

REPR[**shape_offset**(**nof**(N, S))]
= N* REPR[**offset_pad**(**alignment**(S)), **shape_offset**(S))]

REPR[**shape_offset**(top)] = 0

13.2.4 Offset arithmetic

The other OFFSET constructors are given by:

REPR[**offset_zero**(A)] = 0 if ¬ REPR[A].b
REPR[**offset_zero**(A)] = (sd: 0, ad: 0) if REPR[A].b

REPR[**offset_add**(X: OFFSET(A,B), Y: OFFSET(C, D))] = REPR[X] + REPR[Y]
if ¬ REPR[A].b ∧ ¬ REPR[C].b
REPR[**offset_add**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= (sd: REPR[X].sd + REPR[Y].sd, ad: REPR[X].ad + REPR[Y].ad)
if REPR[A].b ∧ REPR[C].b
REPR[**offset_add**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= (sd: REPR[X].sd + REPR[Y], ad:REPR[X].ad)
if REPR[A].b ∧ ¬ REPR[C].b

REPR[**offset_pad**(A, Y: OFFSET(C, D))] = (REPR[Y] + REPR[A].s - 1)/REPR[A].s
if ¬ REPR[A].b ∧ ¬ REPR[C].b
REPR[**offset_pad**(A, Y: OFFSET(C, D))]
= (sd: (REPR[Y] + REPR[A].s - 1)/REPR[A].s, ad: REPR[Y].ad)
if REPR[C].b
REPR[**offset_pad**(A, Y: OFFSET(C, D))]
= (sd: (REPR[Y]+REPR[A].s-1)/REPR[A].s, ad: 0)
if REPR[A].b ∧ ¬ REPR[C].b
REPR[**offset_max**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= Max(REPR[X], REPR[Y])
if ¬ REPR[A].b ∧ ¬ REPR[C].b
REPR[**offset_max**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= (sd: Max(REPR[X].sd, REPR[Y].sd),
ad: Max(REPR[X].a, REPR[Y].ad))
if REPR[A].b ∧ REPR[C].b
REPR[**offset_max**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= (sd: Max(REPR[X].sd, REPR[Y]), ad:REPR[X].ad)
if REPR[A].b ∧ ¬ REPR[C].b
REPR[**offset_max**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= (sd: Max(REPR[Y].sd, REPR[X]), ad: REPR[Y].ad)
if REPR[C].b ∧ ¬ REPR[A].b

REPR[**offset_subtract**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= REPR[X]- REPR[Y]
if ¬ REPR[A].b ∧ ¬ REPR[C].b
REPR[**offset_subtract**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= (sd: REPR[X].sd - REPR[Y].sd, ad:REPR[X].ad - REPR[Y].ad)
if REPR[A].b ∧ REPR[C].b
REPR[**offset_add**(X: OFFSET(A,B), Y: OFFSET(C, D))]
= REPR[X].sd - REPR[Y]
if REPR[A].b ∧ ¬ REPR[C].b
.... and so on.

Unlike the previous one, this model of ALIGNMENTs would reject OFFSETs such as OFFSET({long_variety}, {pointer}) but not OFFSET({pointer}, {long_variety}) since:

REPR [{long_variety} ⊃ {pointer}] = FALSE

but:

REPR [{pointer} ⊃ {long_variety}] = TRUE

This just reflects the fact that there is no way that one can extract a block-address necessary for a pointer from a scalar-block, but since the representation of a pointer includes a scalar displacement, one can always retrieve a scalar from a pointer to a pointer.

14 Conclusion

This commentary is not complete. I have tended to go into considerable detail into aspects which I consider might be unfamiliar and skip over others which occur in most compiling systems. I also have a tendency to say things more than once, albeit in different words; however if something is worth saying, it is worth saying twice.

I shall continue tracking further revisions of the TDF specification in later releases or appendices to this document.