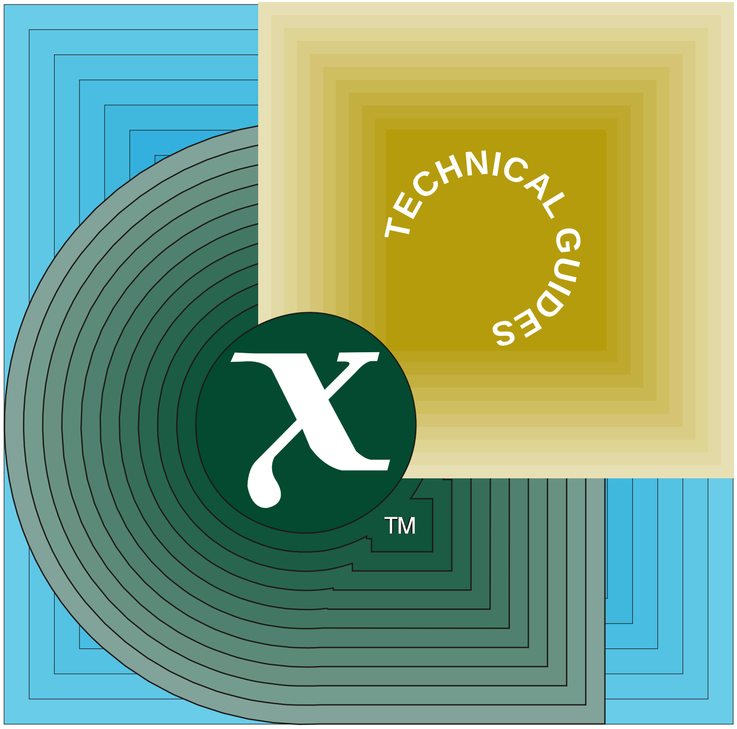
Guide

Architecture Neutral Distribution Format (XANDF)



THE Open GROUP

[This page intentionally left blank]

X/Open Guide

Architecture Neutral Distribution Format (XANDF)

X/Open Company Ltd.

X/Open Guide

Architecture Neutral Distribution Format (XANDF)

ISBN: 1-85912-141-1
X/Open Document Number: G508

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Figures

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done in any one of the following ways:

- anonymous ftp to ftp.xopen.org
- ftpmail (see below)
- reference to the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information using ftpmail, send a message to ftpmail@xopen.org with the following four lines in the body of the message:

```
open
cd pub/Corrigenda
get index
quit
```

This will return the index of publications for which Corrigenda exist. Use the same email address to request a copy of the full corrigendum information following the email instructions.

This Document

This Guide is commentary on the XANDF specification (see reference **XANDF_PS**). The Architecture Neutral Distribution Format (ANDF) is a software porting technology making it possible to develop shrink-wrapped software for open systems, independent of any particular processor architecture. ANDF intends to reduce the effort needed for porting of applications while at the same time making it possible to fully utilise the particular features of a platform.

The XANDF specification defines an integration interface between the two major components of a multi-platform cross-compilation system. The compilation of the source code is turned into a two stage process. In the first stage, the application is transcribed into a format which utilises generalised declarations of the API calls used, together with generalised definitions of data types, constants, and so on. This is the Architecture Neutral Distribution Format and the piece of software producing it is called ANDF producer. XANDF is in fact an abstract algebra.

This format has a value in itself, even if the second phase of the compilation is never entered. It can be examined to determine how portable the code really is, through comparing it against lists of standardised API calls, and issuing warnings when such a search fails. X/Open is currently examining the possibilities offered by these features to its testing programme.

In the second phase of the compilation, the entities generated in the first phase are first linked together and then mapped onto a concrete machine through the use of processor-specific libraries implementing the API calls and data formats. The software performing this task is called ANDF Installer.

The aim of this Guide is elucidate the various constructions of XANDF, giving examples of usages both from the point of view of a producer of XANDF, 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.

Intended Audience

Application writers who wish to examine the portability of their code will be interested in ANDF Producer. Suppliers and Users who want the convenience of running shrink-wrapped applications will be interested in ANDF Installer.

Structure

- **Chapter 1** explains the positioning and purpose of XANDF, and the objectives of this Guide.
- **Chapter 2** explains the use of SORTs and TOKENs.
- **Chapter 3** explains the use of CAPSULEs and UNITs.
- **Chapter 4** explains the use of SHAPEs, ALIGNMENTs and OFFSETs.
- **Chapter 5** describes the definition and usage of Procedures and Locals in XANDF.
- **Chapter 6** describes control flow within Procedures.
- **Chapter 7** describes Values, Variables and Assignments in XANDF.
- **Chapter 8** describes arithmetic operations in XANDF.
- **Chapter 9** describes Constants in XANDF.
- **Chapter 10** describes TOKENs and their use as abstractions for things in libraries and APIs.
- **Chapter 11** describes transformation processes in XANDF.
- **Chapter 12** describes XANDF expansions of Offsets.
- **Chapter 13** describes models of the XANDF algebra.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for commands, keywords, type names, and data structures.
- *Italic* font is used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — command operands, command option-arguments or variable names
  - — environment variables.
- `constant width` font is used for lines quoting code fragments. Where a line of a code fragment is split onto a further line due to text width limitations in this specification, this is denoted by a backslash (\ ) at the end of each line which is split.

Trade Marks

X/Open® is a registered trade mark, and the ‘‘X’’ device is a trade mark, of X/Open Company Limited.

Acknowledgements

The source for this XANDF guide and its companion specification was contributed by the United Kingdom Defence Evaluation and Research Agency (DERA).

Referenced Documents

The following documents are referenced in this Guide:

XANDF_PS
X/Open Preliminary Specification, January 1996, Architecture Neutral Distribution Format (XANDF) (ISBN: 1-85912-146-2, P527).

Chapter 1

Introduction

1.1 Scope and Purpose

The Architecture Neutral Distribution Format (ANDF) is a software porting technology making it possible to develop shrink-wrapped software for open systems, independent of any particular processor architecture. ANDF intends to reduce the effort needed for porting of applications while at the same time making it possible to fully utilise the particular features of a platform.

The XANDF specification defines an integration interface between the two major components of a multi-platform cross-compilation system. The compilation of the source code is turned into a two stage process. In the first stage, the application is transcribed into a format which utilises generalised declarations of the API calls used, together with generalised definitions of data types, constants, and so on. This is the Architecture Neutral Distribution Format and the piece of software producing it is called ANDF producer. XANDF is in fact an abstract algebra.

This format has a value in itself, even if the second phase of the compilation is never entered. It can be examined to determine how portable the code really is, through comparing it against lists of standardised API calls, and issuing warnings when such a search fails. X/Open is currently examining the possibilities offered by these features to its testing programme.

In the second phase of the compilation, the entities generated in the first phase are first linked together and then mapped onto a concrete machine through the use of processor-specific libraries implementing the API calls and data formats. The software performing this task is called ANDF Installer.

While ANDF producers are used by application writers, installers are provided on systems intending to offer the customers the convenience of running shrink-wrapped applications.

1.2 This Guide

This Guide is commentary on the XANDF specification (see reference **XANDF_PS**). If it presents anything which conflicts with the XANDF specification, then the XANDF specification is the authoritative document.

The aim of this Guide is elucidate the various constructions of XANDF, giving examples of usages both from the point of view of a producer of XANDF, 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 when using the XANDF specification.

1.3 Terminology

The following terms are used with a precise meaning which is particular to this document:

compiling
The production of ANDF from some source language.

producing
Same meaning as ‘‘compiling’’.

translating
Making a program for some specific platform from ANDF.

Chapter 2

SORTs and TOKENs

2.1 Overview

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

At the moment, there are 58 different SORTS, from ACCESS to VARIETY, given in Table 2-1 on page 4 and Table 2-2 on page 9. Some of these have familiar analogues in standard language construction as with EXP and TAG above. Others will be less familiar since XANDF must concern itself with issues not normally addressed in language definitions. For example, the process of linking together XANDF programs is at the root of the architecture neutrality of XANDF and so must form an integral part of its definition. On the other hand, XANDF 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 make it much simpler for a computer to produce and analyse XANDF.

The SORTs and constructors in effect form a multi-sorted algebra. There are two principal reasons for choosing this algebraic form of definition:

- It is easy to extend — a new operation on existing constructs simply requires a new constructor.
- The algebraic form is highly amenable to the automatic construction of programs.

Large parts of both XANDF producers and XANDF 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 XANDF. To this extent, one can regard the specification document as a formal description of the free algebra of XANDF SORTs and constructors. Of course, most of the interesting parts of the definition of XANDF lies in the equivalences of parts of XANDF, so this formality only covers the easy part.

Another distinction between the XANDF definition and language syntactic description is that XANDF 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, from the past, Algol-N made a valiant attempt at this. 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.

2.2 Token Applications and First-class SORTs

A new construction is introduced by the SORT TOKEN; the constructors involving TOKENs allow giving an expansion for the TOKEN in terms of other pieces of XANDF, possibly including parameters. We can encapsulate a (possibly parameterised) fragment of XANDF 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** to **variety**). These are the first-class SORTs given in Table 2-1. Each of these have an appropriate **_apply_token** constructor (for example, **exp_apply_token**) give the expansion.

SORT	USAGE	SORT	USAGE
ACCESS	Properties of TAGs	AL_TAG	Name for alignment
ALIGNMENT	Abstraction of data alignment	BITFIELD_VARIETY	Gives no of bits in bit-field with sign
BOOL	true or false	ERROR_TREATMENT	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	PROC PROPS	Properties of calls and definitions of procedures
ROUNDING_MODE	How to round floating point operations	SHAPE	Abstraction of size and representation of values
SIGNED_NAT	Static number of unbounded size.	STRING	Static string of n-bit integers
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

Table 2-1 First Class SORTs

Most of these also have **_cond** constructors, which allows translate time conditional expansion of the SORT.

Every TOKEN has a result SORT (that is, the SORT of its resulting expansion) and before it can be expanded, one must have its parameter SORTs. Thus, a TOKEN can be regarded 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)*
	→ 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 XANDF. 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, and no bits are required to represent **make_unit**; there are about 120 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 in some applications, notably the XANDF 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 XANDF, preceded by the number of bits it occupies. Applications can then skip over 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.

2.3 Token Definitions and Declarations

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**:

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

The *result_sort* is the SORT of the construction of *body*; for example, if *result_sort* is formed from **exp** then *body* would be constructed using the EXP constructors and one would use **exp_apply_token** to give the expansion.

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

The list *tok_params* gives the formal parameters of the definition in terms of TOKFORMALS constructed using **make_tok_formals**:

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 3.2 on page 10). 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)* n 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**:

tok:	TDFINT
signature:	OPTION(STRING)
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 (for example, in **exp_apply_token**) will be given by **make_token** (*tok*). Once again, a BITSTREAM is used to encapsulate the TOKEN_DEFN.

The significance of the *signature* parameter is discussed in Section 3.3.2 on page 16.

Often, one wishes a token without giving its definition — the definition could, for example, be platform-dependent. A TOKDEC introduces such a token using **make_tokdec**:

tok:	TDFINT
signature:	OPTION(STRING)
s:	SORTNAME
	→ TOKDEC

Here the SORTNAME, *s* is given by

token:	
result:	SORTNAME
params:	LIST(SORTNAME)
	→ SORTNAME

which gives the result and parameter SORTs of *tok*.

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**:

tdef: BITSTREAM TOKEN_DEFN
→ TOKEN

2.4 A Simple Use of a TOKEN

The crucial use of TOKENs in XANDF is to provide abstractions of APIs (see Chapter 10 on page 55) but they are also used as shorthand for commonly occurring constructions. For example, given the XANDF constructor **plus**, mentioned above, we could define a plus with only two EXP parameters more suitable to C by using the **wrap** constructor as the ERROR_TREATMENT:

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

2.5 Second Class SORTs

Second class SORTs (given in Table 2-2 on page 9) cannot be TOKENised. These are the syntactic units of XANDF which the user cannot extend; the user can only produce them using the constructors defined in core-XANDF.

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 XANDF. However, it is for seen that LIST constructors would be highly desirable and there will probably extensions to XANDF 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.

SORT	USAGE	SORT	USAGE
AL_TAGDEF	Alignment name definition	AL_TAG DEF_PROPS	Body of UNIT containing AL_TAGDEFs
BITSTREAM	Encapsulation of a bit encoding	BYTE STREAM	Encapsulation of a byte encoding
CALLEES	Actual callee parameters	CAPSULE	Independent piece of XANDF program
CAPSULE_LINK	No and kind of linkable entities in CAPSULE	CASELIM	Bounds in case constructor

SORT	USAGE	SORT	USAGE
ERROR_CODE	Encoding for exceptions	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; may have SORTNAME later.
OTAGEXP	Describes a formal parameter	PROPS	Program info in a UNIT
SLIST(AUX)	List of AUX SORTs; will not have SORTNAME later	SORTNAME	SORT which can be parame ter of TOKEN
TAGACC	Used in constructing proc formals	TAGDEC	Declaration of TAG at UNIT level
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	XANDF encoding for a boolean	TDFIDENT	XANDF encoding of a byte string
TDFINT	XANDF encoding of an integer	TDFSTRING	XANDF 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	TOKFORMALS	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 XANDF	VERSION_PROPS	Body of UNIT containing version information

Table 2-2 Sorts Without SORTNAMEs

Chapter 3

CAPSULEs and UNITs

3.1 CAPSULE

A CAPSULE is typically the result of a single compilation — one could regard it as being the XANDF 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.

3.2 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** is represented graphically in Figure 3-1.

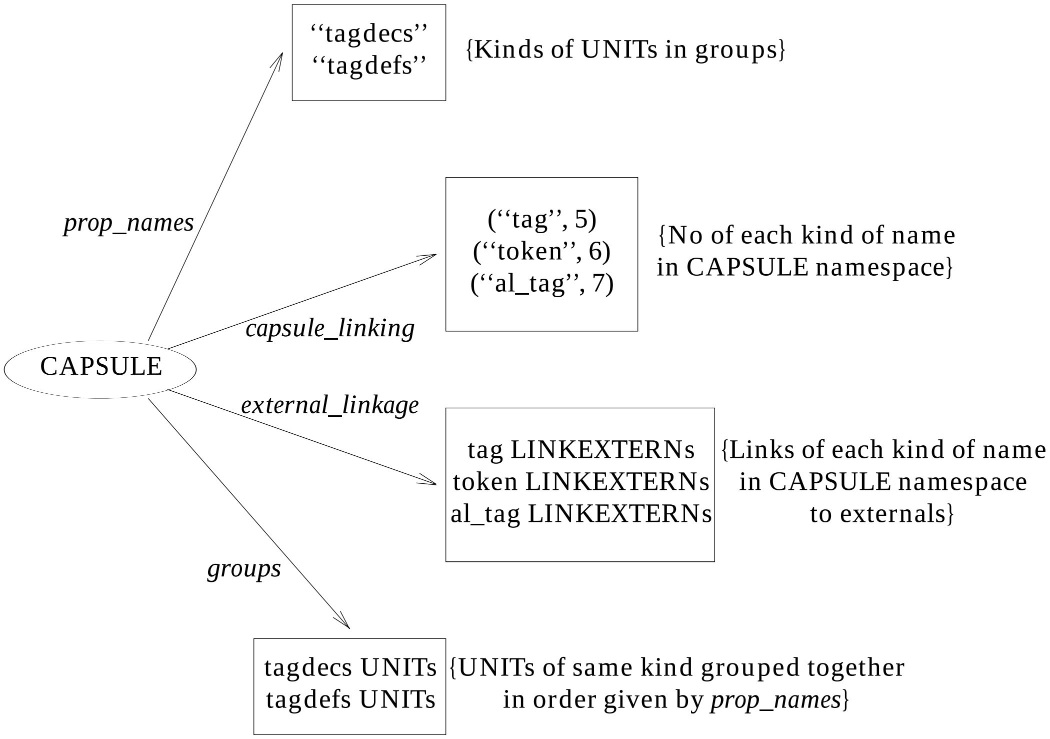


Figure 3-1 An Example of a Capsule

Figure 3-1 gives an example of a CAPSULE using the same components as in the following description.

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 3.2.2 on page 12). There are several different kinds of names in XANDF 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-XANDF, namely **tag**, **token** and **al_tag**. Those names in a **tag** name-space generally correspond to identifiers in normal programs and so this Guide uses 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-XANDF, there would be three entries in the list introducing limits using **make_capsule_link** 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.

3.2.1 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** applied to an 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 is itself formed from an SLIST of LINKEXTERNs given by **make_extern_link**. 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 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 a 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.

3.2.2 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 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-XANDF only uses those named **‘‘tld’’**, **‘‘al_tagdefs’’**, **‘‘tagdecs’’**, **‘‘tagdefs’’**, **‘‘tokdecs’’** and **‘‘tokdefs’’**. These names[2] 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.

[2] The **‘‘tld’’** UNITs gives usage information for names to aid the linker, tld, to discover which names have definitions and some usage information. The C producer also optionally constructs diagnostics UNITs (to give run-time diagnostic information).

3.2.3 make_unit

Consider the construction of UNITs using **make_unit**, as in Figure 3-2.

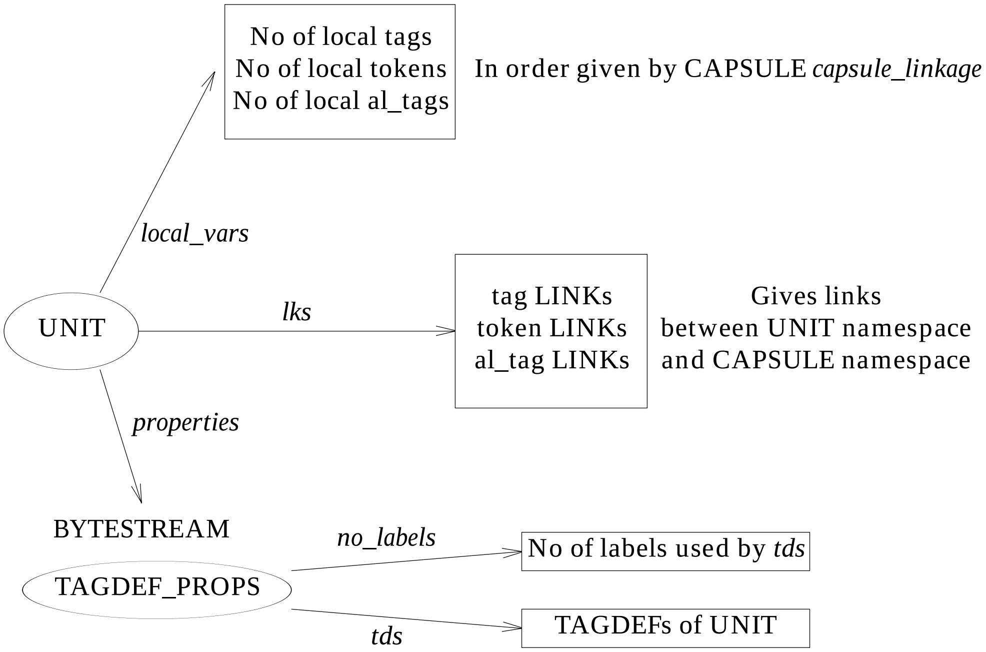


Figure 3-2 An Example of a tagdef Unit

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 3.2 on page 10. 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, and so on.. 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.

3.2.4 LINK

Connections between the CAPSULE name-spaces and the UNIT name-spaces are made by LINKs 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 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.

3.3 Definitions and Declarations

The encoding in the *properties:BYTSTREAM* parameter of a UNIT is a PROPS, for which there are five constructors corresponding to the kinds of UNITs in core-XANDF, **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). For tokens, the ‘‘type’’ is a SORTNAME constructed from the SORTNAMEs of the parameters and result of the TOKEN using **token**:

params:	LIST(SORTNAME)
result:	SORTNAME
	→ SORTNAME

Taking **make_tagdefs** as a paradigm for PROPS, we have:

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 used in the TAGDEFs. Each TAGDEF 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 values which can be solved at translate time, load time or as unordered dynamic initialisations.

There are three constructors for TAGDEFs, each with slightly different properties. The simplest is **make_id_tagdef**:

t:	TDFINT
signature:	OPTION(STRING)
e:	EXP *x*
	→ TAGDEF

Here, *t* is the tag name and the evaluation of *e* will be the value of SHAPE *x* of an **obtain_tag***(t)* in an EXP. Note that *t* is not a variable; the value of **obtain_tag***(t)* will be invariant. The *signature* parameter gives a STRING which may be used as an name for the tag, external to XANDF and also as a check introduced by the producer that a **tagdef** and its corresponding **tagdec** have the same notion of the language-specific type of the tag.

The two other constructors for TAGDEF, **make_var_tagdef** and **common_tagdef**, both define variable tags and have the same signature:

t:	TDFINT
opt_access:	OPTION(ACCESS)
signature:	OPTION(STRING)
e:	EXP *x*
	→ TAGDEF

Once again *t* is tag name but now *e* is initialisation of the variable *t*. A use of **obtain_tag** (*t*) will give a pointer to the variable (of SHAPE POINTER x), rather than its contents[3]. There can only be one **make_var_tagdef** of a given tag in a program, but there may be more than one **common_tagdef**, possibly with different initialisations; however these initialisations must overlap consistently just as in common blocks in FORTRAN.

The ACCESS parameter gives various properties required for the tag being defined.

The initialisation EXPs of TAGDEFs will be evaluated before the ‘‘main’’ program is started. An initialisation EXP must either be a constant (in the sense of Chapter 9), or reduce to (either directly or by token or *_cond* expansions) to an **initial_value**:

init:	EXP *s*
	→ EXP *s*

The translator will arrange that *init* will be evaluated once only before any procedure application, other than those themselves involved in *initial_values*, but after any constant initialisations. The order of evaluation of different *initial_values* is arbitrary.

3.3.1 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). They can be regarded 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 is constructed using either **make_id_tagdec**, **make_var_tagdec** or **common_tagdec**, all with the same form:

t_intro:	TDFINT
acc:	OPTION(ACCESS)
signature:	OPTION(STRING)
x:	SHAPE
	→ TAGDEC

Here the tagname is given by *t_intro*; the SHAPE *x* will define the space and alignment 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; the kinds of properties envisaged in discussing local declarations are explained later.

[3] There is a similar distinction between tags introduced to be locals of a procedure using **identify** and **variable**.

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 XANDF 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. For example, 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 XANDF linking. Other languages have more stringent requirements. For example, C++ requires dynamic initialisation of globals, using **initial_value**. As the only runnable code in XANDF is in procedures, C++ would probably require an additional linking phase to construct a ‘‘main’’ procedure which calls the initialisation procedures of each CAPSULE involved if the system linker did not provide suitable C++ linking.

3.3.2 Declaration and Definition Signatures

The *signature* arguments of TAGDEFs and TAGDECs are designed to allow a measure of cross-UNIT checking when linking independently compiled CAPSULEs. Suppose that we have a tag, *t*, used in one CAPSULE and defined in another; the first CAPSULE would have to have a TAGDEC for *t* whose TAGDEF is in the second. The *signature* STRING of both could be arranged to represent the language-specific type of *t* as understood at compilation-time. Clearly, when the CAPSULEs are linked the types must be identical and hence their STRING representation must be the same — a translator will reject any attempt to link definitions and declarations of the same object with different signatures.

Similar considerations apply to TOKDEFs and TOKDECs; the ‘‘type’’ of a TOKEN may not have any familiar analogue in most HLLs, but the principle remains the same.

3.3.3 STRING

The SORT STRING is used in various constructs other than declarations and definitions. It is a first-class SORT with **string_apply_token** and **string_cond**. A primitive STRING is constructed from a TDFSTRING(k,n) which is an encoding of n integers, each of k bits, using **make_string**:

arg: TDFSTRING(k, n)
→ STRING(k, n)

STRINGs may be concatenated using **concat_string:**

arg1: STRING(k, n)
arg2: STRING(k,m)
→ STRING(k, n+m)

Being able to compose strings, including token applications, and so on, means that late-binding is possible in signature checking in definitions and declarations. This late-binding means that the representation of platform-dependent HLL types need only be fully expanded at install-time and hence the types could be expressed in their representational form on the specific platform.

Chapter 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 XANDF analogue of the type of value is its SHAPE. 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** to **top**, as shown in Table 4-1.

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.

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 ⊇ 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 ⊇ a2.	{offset}
POINTER(a);	ALIGNMENT	Pointers in memory	{pointer}
PROC		Procedure values	{proc}
TOP		No value; e.g., result of assign	{ }

Table 4-1 SHAPE Basic Constructors

In the XANDF specification of EXPs, all EXPs in constructor signatures are 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*). It should not be assumed that XANDF is strongly-typed by its SHAPEs, but 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 an XANDF 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** and **integer_test**. A BOTTOM SHAPE is produced by an expression which will leave the current flow of control, for example **goto**. 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 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

Integer values in XANDF have shape INTEGER(*v*) where *v* is of SORT VARIETY. The constructor for this SHAPE is *integer* with a VARIETY parameter. The basic constructor for VARIETY is *var_limits* 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. However, most current translators do not handle integers of more than the maximum given naturally by the target architecture; this will be rectified in due course.

The other way of constructing a VARIETY is to specify the number of bits required for its twos-complement representation using **var_width**:

signed_width:	BOOL
width:	NAT
	→ VARIETY

4.1.3 FLOATING and Complex

Similarly, floating point and complex numbers have shape FLOATING qualified by a FLOATING_VARIETY.

A FLOATING_VARIETY for a real number is constructed using **fvar_parms**:

base:	NAT
mantissa_digits:	NAT
minimum_exponent:	NAT
maximum_exponent:	NAT

→ FLOATING_VARIETY

A FLOATING_VARIETY 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.

Complex numbers have a floating variety constructed by **complex_parms** which has the the same signature as **fvar_parms**. The representation of these numbers is likely to be a pair of real numbers each defined as if by **fvar_parms** with the same arguments. The real and imaginary parts of of a complex number can be extracted using **real_part** and **imaginary_part**; these could have been injected into the complex number using **make_complex** or any of the complex operations. Many translators will simply transform complex numbers into COMPOUNDs consisting of two floating point numbers, transforming the complex operations into floating point operations on the fields.

4.1.4 BITFIELD

A number of contiguous bits have shape BITFIELD, qualified by a BITFIELD_VARIETY 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 or 64 on the number of bits.

4.1.5 PROC

The representational SHAPEs of procedure values is given by PROC with constructor **proc**.

4.1.6 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**, **nof**, **offset** and **pointer**. Before describing these (see Section 4.4 on page 25, it is necessary to explain 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 XANDF — 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, for example, 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 XANDF abstraction of an alignment has SORT ALIGNMENT. The constructor **unite_alignments** 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** 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 4-1 on page 19. Those for PROC, all OFFSETs and all POINTERs are constants regardless of any SHAPE qualifiers. Each of the INTEGER 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).

The alignment of a procedure parameter clearly has to include the alignment of its SHAPE; however, most ABIs will mandate a greater alignment for some SHAPEs, for example the alignment of a byte parameter is usually defined to be on a 32-bit rather than an 8-bit boundary. The constructor **parameter_alignment** gives the ALIGNMENT of a parameter of given SHAPE.

4.2.2 Special Alignments

There are several other special ALIGNMENTs.

The alignment of a code address is { *code* } given by **code_alignment** ; this will be the alignment of a pointer given by **make_local_lv** giving the value of a label.

The other special ALIGNMENTs are considered to include all of the others, but remain distinct. They are all concerned with offsets and pointers relevant to procedure frames, procedure parameters and local allocations and are collectively known as frame alignments. These frame alignments differ from the normal alignments in that their mapping to a given architecture is rather more than just saying that it describes some n-bit boundary. For example, **alloca_alignment** describes the alignment of dynamic space produced by **local_alloc** (roughly the C **alloca**). Now, an ABI could specify that the alloca space is a stack disjoint from the normal procedure stack; thus manipulations of space at **alloca_alignment** may involve different code to space generated in other ways.

Similar considerations apply to the other special alignments, **callees_alignment** *(b)*, **callers_alignment** *(b)* and **locals_alignment.** The first two give the alignments of the bases of the two different parameter spaces in procedures (see Chapter 5 on page 29) and **locals_alignment** gives the alignment of the base of locally declared tags within a procedure. The exact interpretation of these depends on how the frame stack is defined in the target ABI, , for example, does the stack grow downwards or upwards?

The final special alignment is **var_param_alignment.** This describes the alignment of a special kind of parameter to a procedure which can be of arbitrary length (see Section 5.1.2 on page 31).

4.2.3 AL_TAG, make_al_tagdef

Alignments can also be named as AL_TAGs using **make_al_tagdef**. There is no corresponding **make_al_tagdec** since AL_TAGs are implicitly declared by their constructor, **make_al_tag**. 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 XANDF SHAPE for a pointer is POINTER qualified by the ALIGNMENT of the object pointed to. The constructor **pointer** uses this alignment to make a POINTER SHAPE.

4.3.1 OFFSET

Expressions which give sizes or offsets in XANDF 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** uses two such alignments to make an OFFSET SHAPE. However, many instances of offsets will be produced implicitly by the offset arithmetic, for example, **offset_pad**:

a:	ALIGNMENT
arg1:	EXP OFFSET(z,t)
	→ 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 (that is, 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
    → 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*), that is, the alignment of an array is just the alignment of its elements.

The other compound SHAPEs are produced using compound:

```
sz:  EXP OFFSET(x,y)
     → 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**, **offset_zero** and **shape_offset** are used together with **offset_pad** to implement (among other things) 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** or **add_to_ptr**. 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 XANDF

The SHAPE of a C structure consisting of a **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 13.1 on page 69). 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 Chapter 12.

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** 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** and **offset_max** having properties like:

shape_offset$(S) \geq$ **offset_zero**(alignment(S))
$A \geq B$ iff **offset_max**$(A,B) = A$
offset_add$(A, B) \geq A$ where $B \geq$ **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. The reasons why XANDF does not simply use integers for offsets, instead of introducing the rather complex OFFSET SHAPE, are two-fold:

- Firstly, 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 XANDF 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.
- Secondly, 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* }) (for example, a pointer to a structure containing both integers and pointers) could have two components, 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 Chapter 13.

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[4]. 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 in the case of the most commonly used ones.

XANDF does not allow one to construct a pointer of SHAPE POINTER(*b*) where *b* consists entirely of bitfield alignments. This relieves the translators of the burden of doing general bit-addressing. Of course, this simply shifts the burden to the producer. If the high level language requires to construct a pointer to an arbitrary bit position, then the producer is required to represent such a pointer as a pair consisting of pointer to some alignment including the required bitfield and an offset from this alignment to the bitfield. For example, Ada may require the construction of a pointer to a boolean for use as the parameter to a procedure; the SHAPE of the representation of this Ada pointer could be a COMPOUND formed from a POINTER({ *x*, *b* }) and an OFFSET({ *x*, *b* }, *b*) where *b* is the alignment given by a 1 bit alignment. To access the boolean, the producer would use the elements of this pair as arguments to **bitfield_assign** and **bitfield_contents**.

[4] Note that it 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. For more on similar limitations of bitfields in XANDF see ‘‘Assigning and Extracting Bitfields’’.

Chapter 5

Procedures and Locals

All procedures in XANDF 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.

All executable code in XANDF will arise from an EXP PROC made by either **make_proc** or **make_general_proc**. They differ in their treatment of how space for the actual parameters of a call is managed; in particular, is it the caller or the callee which deallocates the parameter space?

With **make_proc,** this management is conceptually done by the caller at an **apply_proc**; that is, the normal C situation. This suffers from the limitation that tail-calls of procedures are then only possible in restricted circumstances (for example, the space for the parameters of the tail-call must be capable of being included in caller's parameters) and could only be implemented as an optimisation within a translator. A producer could not predict these circumstances in a machine independent manner, whether or not it knew that a tail-call was valid.

An alternative would be to make the management of parameter space the responsibility of the called procedure. Rather than do this, **make_general_proc** (and **apply_general_proc**) splits the parameters into two sets, one whose allocation is the responsibility of the caller and the other whose allocation is dealt with by the callee. This allows an explicit **tail_call** to be made to a procedure with new callee parameters; the caller parameters for the **tail_call** will be the same as (or some initial subset of) the caller parameters of the procedure containing the **tail_call**.

A further refinement of **make_general_proc** is to allow access to the caller parameter space in a postlude at the call of the procedure using an **apply_general_proc.** This allows simple implementations of Ada out_parameters, or more generally, multiple results of procedures.

5.1 Traditional Procedures

5.1.1 make_proc and apply_proc

The **make_proc** constructor has signature:

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 usually given by some **return** construction within *body*. The basic model is that space will be provided to copy actual parameters (into space supplied by some **apply_proc**) by value into these formals and the body will treat this space effectively as local variables.

Each straightforward formal parameter is introduced by an auxiliary SORT TAGSHACC using **make_tagshacc**:

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. For more insight on the meaning of *opt_access* and the ramifications of the scope and extent of TAGs involved in conjunction with local declarations, see Section 5.3 on page 33.

Procedures, whether defined by **make_proc** or **make_general_proc**, will usually terminate and deliver its result with a **return**:

arg1:	EXP
	→ EXP BOTTOM

Here *x* must be identical to the *result_shape* of the call of the procedure There may be several returns in *body*; and the SHAPE *x* in each will be the same. 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 XANDF a dummy value must be provided at the return; for example **make_value** (*result_shape*)

Note that the *body* has SHAPE bottom since all possible terminations to a procedure have SHAPE BOTTOM.

Procedures defined by **make_proc** are called using **apply_proc**:

result_shape:	SHAPE
arg1:	EXP PROC
arg2:	LIST(EXP)
varparam:	OPTION(EXP)
	→ EXP *result_shape*

Here *arg1* is the procedure to be called and *arg2* gives the actual parameters. There must be at least as many actual parameters as given (with the same SHAPE) in the *params_intro* of the corresponding **make_proc** for *arg1*. The values of *arg2* will be copied into space managed by caller.

The SHAPE of the result of the call is given by *result_shape* which must be identical to the *result_shape* of the **make_proc**.

5.1.2 vartag, varparam

Use of the *var_intro* OPTION in **make_proc** and the corresponding *varparam* in **apply_proc** allows one to have a parameter of any SHAPE, possibly differing from call to call where the actual SHAPE can be deduced in some way by the body of the **make_proc**. One supplies an extra actual parameter, *varparam,* which usually would be a structure grouping some set of values. The body of the procedure can then access these values using the pointer given by the TAG *var_intro*, using **add_to_ptr** with some computed offsets to pick out the individual fields.

This is a slightly different method of giving a variable number of parameters to a procedure, rather than simply giving more actuals than formals. The principle difference is in the alignment of the components of *varparam*; these will be laid out according to the default padding defined by the component shapes. In most ABIs, this padding is usually different to the way parameters are laid out; for example, character parameters are generally padded out to a full word. Thus a sequence of parameters of given shape has a different layout in store to the same sequence of shapes in a structure. If one wished to pass an arbitrary structure to a procedure, one would use the *varparam* option rather passing the fields individually as extra actual parameters.

5.2 General Procedures

5.2.1 make_general_proc, apply_general_proc

A **make_general_proc** has signature:

result_shape:	SHAPE
prcprops:	OPTION(PROCPROPS)
caller_intro:	LIST(TAGSHACC)
callee_inro:	LIST(TAGSHACC)
body:	EXP BOTTOM
	→ EXP PROC

Here the formal parameters are split into two sets, *caller_intro* and *callee_intro,* each given by a list of TAGSHACCs just as in **make_proc**. The distinction between the two sets is that the **make_general_proc** is responsible for deallocating any space required for the callee parameter set; this really only becomes obvious at uses of **tail_call** within *body*.

The *result_shape* and *body* have the same general properties as in **make_proc.** In addition *prcprops* gives other information both about body and the way that that the procedure is called. PROCPROPS are a set drawn from **check_stack**, **inline**, **no_long_jump_dest**, **untidy**, **var_callees** and **var_callers**. The set is composed using **add_procprops**. The PROCPROPS **no_long_jump_dest** is a property of *body* only; it indicates that none of the labels within *body* will be the target of a **long_jump** construct. The other properties should also be given consistently at all calls of the procedure; these are discussed in Section 5.2.3.

A procedure, *p,* constructed by **make_general_proc** is called using **apply_general_proc** :

result_shape:	SHAPE
prcprops:	OPTION(PROCPROPS)
p:	EXP PROC

caller_params: LIST(OTAGEXP)
callee_params: CALLEES
postlude: EXP TOP
→ EXP *result_shape*

The actual caller parameters are given by *caller_params* as a list of OTAGEXPs constructed using **make_otagexp**:

tgopt: OPTION(TAG*x*)
e: EXP*x*
→ OCTAGEXP

Here, *e* is the value of the parameter and *tgopt*, if present, is a TAG which will bound to the final value of the parameter (after *body* is evaluated) in the postlude expression of the **apply_general_proc**[5]. Clearly, this allows one to use a caller parameter as an extra result of the procedure, for example, as in Ada out-parameters.

The actual *callee_params* may be constructed in three different ways. The usual method is to use **make_callee_list**, giving a list of actual EXP parameters, corresponding to the *caller_intro* list in the obvious way. The constructor **same_callees** allows one to use the callees of the current procedure as the callees of the call; this, of course, assumes that the formals of the current procedure are compatible with the formals required for the call. The final method allows one to construct a dynamically sized set of CALLEES: **make_dynamic_callees** takes a pointer and a size (expressed as an OFFSET) to make the CALLEES; this will be used in conjunction with a *var_callees* PROCPROPS (see Section 5.2.3).

Some procedures can be expressed using either **make_proc** or **make_general_proc.** For example:

```
make_proc(S, L, empty, B) = make_general_proc(S, var_callers, L, empty, B)
```

5.2.2 tail_call

Often the result of a procedure, *f*, is simply given by the call of another (or the same) procedure, *g*. In appropriate circumstances, the same stack space can be used for the call of *g* as the call of *f*. This can be particularly important where heavily recursive routines are involved. Some languages even use tail recursion as the preferred method of looping.

One condition for such a tail call to be applicable is knowing that *g* does not require any pointers to locals of *f*; this is often implicit in the language involved. Equally important is that the action on the return from *f* is indistinguishable from the return from *g*. For example, if it were the callers responsibility to pop the the space for the parameters on return from a call, then the tail call of *g* would only work if *g* had the same parameter space as *f*.

This is the justification for splitting the parameter set of a general proc. It is (at least conceptually) the caller's responsibility for popping the caller-parameters only — the callee parameters are dealt with by the procedure itself. Hence we can define **tail_call** which uses the same caller parameters but a different set of callee parameters:

[5] If a formal parameter is to be used in this way, it should be marked as having out_par ACCESS in its corresponding TAGSHACC in *callers_intro*.

prcprops:	OPTION(PROCPROPS)
p:	EXP PROC
callee_params:	CALLEES
	→ EXP BOTTOM

The procedure *p* will be called with the same caller parameters as the current procedure and the new *callee_params* and return to the call site of the current procedure. Semantically, if S is the return SHAPE of the current procedure, and L is its caller-parameters:

```
tail_call(P, p, C) = return(apply_general_proc(S, P, p, L, C, make_top( )))
```

However an implementation is expected to conserve stack by using the same space for the call of p as the current procedure.

5.2.3 PROCPROPS

The presence of *var_callees* (or *var_callers*) means that the procedure can be called with more actual callee (or caller) parameters than are indicated in *callee_intro* (or *caller_intro*). These extra parameters would be accessed within body using offset calculations with respect to the named parameters. The offsets should be calculated using **parameter_alignment** to give the packing of the parameter packs.

The presence of *untidy* means that *body* may be terminated by an *untidy_return.* This returns the result of the procedure as in *return*, but the lifetime of the local space of the procedure is extended (in practice this is performed by not returning the stack to its original value at the call). A procedure containing an *untidy_return* is a generalisation of a **local_alloc** (see Section 5.3.4). For example, the procedure could do some complicated local allocation (a triangular array, say) and untidily return a pointer to it so that the space is still valid in the calling procedure. The space will remain valid for the lifetime of the calling procedure unless some **local_free** is called within it, just as if the space had been generated by a **local_alloc** in the calling procedure.

The presence of *inline* is just a hint to the translator that the procedure body is a good candidate for inlining at the call.

The presence of **check_stack** means that the static stack requirements of the procedure will be checked on entry to see that they do not exceed the limits imposed by **set_stack_limit**; if they are exceeded an XANDF exception with ERROR_CODE *stack_overflow* (see Section 6.3.0 on page 42) will be raised.

5.3 Defining and Using Locals

5.3.1 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 XANDF transformations. The simpler constructor is **identify**:

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

The other kind of local definition is **variable**:

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**, which will produce some value with SHAPE given by its parameter.

5.3.2 ACCESS

The ACCESS SORT given in tag declarations is a way of describing a list of properties to be associated with the tag. They are basically divided into two classes, one which describes global properties of the tag with respect to the model for locals and the other which gives hints on how the value will be used. Any of these can be combined using **add_access**.

Locals Model

At the moment there are just four possibilities in the first class of ACCESS constructors. They are **standard_access** (the default) , **visible**, **out_par** and **long_jump_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 either from the start of the frame or some pointer within it.

Indeed, if the ACCESS *opt_access* parameter in a TAG definition is produced by **visible**, 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**, which is the standard way of recovering the value bound to the TAG. The principal way that this could happen within XANDF is by the combined use of **env_offset** to give the offset and **current_env** to give a pointer to the current frame (see Section 5.3.3).

The **out_par** ACCESS is only applicable to caller parameters of procedures; it indicates that the value of the TAG concerned will accessed by the postlude part of an **apply_general_proc.** Hence, the value of the parameter must be accessible after the call. Usually this will be on the stack in the callers frame.

The **long_jump_access** flag is used to indicate that the tag must be available after a **long_jump**. 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 **contents** or the left-hand operands of **assigns**, most ABIs would allow the tag to be placed in a register. It is not necessarily a requirement to generate a pointer value if it can be subsumed by the operations available.

Access Hints

A variable tag with ACCESS **constant** is a write-once value; once it is initialised, the variable will always contain the initialisation. In other words, the tag is a pointer to a constant value. Translators can use this information to apply various optimisations.

A POINTER tag with ACCESS **no_other_read** or **no_other_write** is asserting that there are no ‘‘aliased’’ accesses to the contents of the pointer. For example, when applied to a parameter of a procedure, it is saying that the original pointer of the tag is distinct from any other tags used (reading/writing) in the lifetime of the tag. These other tags could either be further parameters of the procedure or globals. Clearly, this is useful for describing the limitations imposed by Fortran parameters, for example.

5.3.3 current_env, env_offset

The constructor **current_env** gives a pointer to the current procedure frame of SHAPE POINTER(*fa*) where *fa* is depends on how the procedure was defined and will be some set of the special frame ALIGNMENTs. This set will always include **locals_alignment** — the alignment of any locals defined within the procedure. If the procedure has any caller parameters, the set will also include **callers_alignment**(*b*) where *b* indicates whether there can be a variable number of them. Similarly for callee parameters.

Offsets from the **current_env** of a procedure to a tag declared in the procedure are constructed by **env_offset**:

```
fa:  ALIGNMENT
 y:  ALIGNMENT
 t:  TAGx
     → EXP OFFSET(fa,y)
```

The frame ALIGNMENT *fa* will be the appropriate one for the TAG *t*; that is, if *t* is a local then the *fa* will be **locals_alignment**; if *t* is a caller parameter, *fa* will be **callers_alignment (***b*); if *t* is a callee parameter, *fa* will be **callees_alignment(***b*). The alignment *y* will be the alignment of the initialisation of *t*.

The offset arithmetic operations allow one to access the values of tags non-locally using values derived from **current_env** and **env_offset.** They are effectively defined by the following identities:

- If TAG *t* is derived from a variable definition:

```
add_to_ptr(current_env(),  env_offset(locals_alignment,  A, t))
    = obtain_tag(t)
```

- If TAG *t* is derived from an identify definition:

```
contents(S,add_to_ptr(current_env(),env_offset(locals_alignment,A,t)  ))
    = obtain_tag(t)
```

- If TAG *t* is derived from a caller parameter:

```
add_to_ptr(current_env(),  env_offset(callers_alignment(b),  A, t))
    = obtain_tag(t)
```

- If TAG *t* is derived from a callee parameter:

```
add_to_ptr(current_env(),  env_offset(callees_alignment(b),  A, 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 within the enclosing UNIT, 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.

5.3.4 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. XANDF also allows one to produce dynamically sized local objects which are conceptually part of the frame. These are produced by **local_alloc**:

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**( ); this reduces the size of the current frame to its static size.

The other constructor is **local_free** which 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**. 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

The use of a procedure with an *untidy_return* is just a generalisation of the idea of **local_alloc** and the space made available by its use can be freed in the same way as normal local allocations. Of course, given that it could be the result of the procedure it can be structured in an arbitrarily complicated way.

5.4 Heap Storage

At the moment, there are no explicit constructors for creating dynamic off-stack storage in XANDF. 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**.

Chapter 6

Control Flow within Procedures

6.1 Unconditional Flow

6.1.1 Sequence

To perform a sequential set of operations in XANDF, one uses the constructor sequence:

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 wherever the text states ‘‘something is evaluated and then...’’ . You will see this kind of statement in definitions of local variables in Section 5.3 on page 33, and in the controlling parts of the conditional constructions below.

6.2 Conditional Flow

6.2.1 labelled, make_label

All simple changes of flow of control within an XANDF 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**, 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 *places* is labelled by the corresponding LABEL in *placelabs_intro*. These LABELs are constructed by **make_label** 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 *places*, 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, return_to_label

The unconditional **goto** 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** for later use by an indirect jump such as **goto_local_lv**. 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** 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.

A slightly shorter long jump is given by **return_to_label**. Here, the destination of the jump must a label value in the calling procedure. Usually this value would be passed as parameter of the call to provide an alternative exit to the procedure.

6.2.3 integer_test, NTEST

Conditional branching is provided by the various **_test** constructors, one for each primitive SHAPE except BIT FIELD. **integer_test** is used as the paradigm for them all:

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

The NTEST *nt* chooses a dyadic test (for example, =, ≥, <, and so on) 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 (for example, **proc_test**) while others are redundant for some **_test**s. For example, **not_greater_than** is the same as **less_than_or_equal** 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:**

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

Each CASELIM is constructed using **make_caselim:**

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:**

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 XANDF.

A simple loop can be expressed using **repeat**:

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 continuing or breaking the loop are included in *body* and require no special constructions.

6.3 Exceptional Flow

A further way of changing the flow of control in an XANDF program is by means of exceptions. XANDF exceptions currently arise from three sources:

- overflow in arithmetic operations with **trap** ERROR_TREATMENT (see Section 8.1.1 on page 48)
- an attempt to access a value via a nil pointer using **assign_with_mode**, **contents_with_mode** or **move_some** (see Section 7.2.1 on page 44)
- a stack overflow on procedure entry with PROCPROPS **check_stack** (see Section 5.2.3 on page 33), or a **local_alloc_check**.

Each of these exceptions have an ERROR_CODE ascribed to them, respectively *overflow*, *nil_access* and *stack_overflow*. Each ERROR_CODE can be made into a distinct NAT by means of the constructor **error_val**. These NATs could be used, for example, to discriminate the different kinds of errors using a case construction.

When one of these exceptions is raised, the translator will arrange that a TOKEN, ˜Throw, is called with the appropriate **error_val** NAT as its (sole) parameter. Given that every language has a different way of both characterising and handling exceptions, the exact expansion of ˜Throw must be given by the producer for that language — usually it will involve doing a **long_jump** to some label specifying a signal handler and translating the ERROR_CODE into its language-specific representation.

The expansion of ˜Throw forms part of the language specific environment required for the translation of XANDF derived from the language, just as the exact shape of FILE must be given for the translation of C.

Chapter 7

Values, Variables and Assignments

TAGs in XANDF 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 (that is, 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 **content**s 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 XANDF is done using **assign** :

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 XANDF in translators and other tools (see Chapter 11).

7.2.1 TRANSFER_MODE Operations

The TRANSFER_MODE operations allow one to do assignment and contents operations with various qualifiers to control how the access is done in a more detailed manner than the standard **contents** and **assign** operations.

For example, 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**:

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. 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.

If the TRANSFER_MODE **trap_on_nil** is present and *arg1* is a nil pointer, an XANDF exception with ERROR_CODE *nil_access* is raised.

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 constructors *md* are **trap_on_nil** and **volatile**. The latter 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**.

7.3 Assigning and Extracting Bitfields

Since pointers to bits are forbidden, two special operations are provided to extract and assign bitfields. These require the use of a pointer value and a bitfield offset from the pointer. The signature of **bitfield_contents** which extracts a bitfield in this manner is:

v:	BITFIELD_VARIETY
arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
	→ EXP bitfield(*v*)

Here *arg1* is a pointer to an alignment *x* which includes *v*, the required bitfield alignment. In practice, *x* must include an INTEGER VARIETY whose representation can contain the entire bitfield. Thus on a standard architecture, if *v* is a 15-bit bitfield, x must include at least a 16-bit integer variety; a 27-bit field would require a 32-bit integer variety, and so on. Indeed the constraint is stronger than this since there must be an integer variety, accessible from *arg1*, which entirely contains the bitfield.

This constraint means that producers cannot expect that arbitrary bitfield lengths can be accommodated without extra padding. Clearly, it also means that the maximum bitfield length possible is the maximum size of integer variety that can be implemented on the translator concerned (this is defined to be at least 32). On standard architectures, the producer can expect that an array of bitfields of length 2^n will be packed without padding; this, of course, includes arrays of booleans. For structures of several different bitfields, the producer can be sure of no extra padding bits if the total number of bits involved is less than or equal to 32, and similarly so if the producer can subdivide the bitfields so that each of the subdivisions (except the last) is exactly equal to 32 and the last is less than or equal to 32. This could be relaxed to 64 bits if the translator deals with 64-bit integer varieties, but would require that conditional XANDF is produced to maintain portability to 32-bit platforms — and on these platforms the assurance of close packing would be lost.

Since a producer is ignorant of the exact representational varieties used by a translator, the onus is on the translator writer to provide standard tokens which can be used by a producer to achieve both optimum packing of bit fields and minimum alignments for COMPOUNDs containing them (see Section 12.1 on page 66). These tokens would allow one to construct an offset of the form OFFSET(*a*,*b*) (where *b* is some bitfield alignment and *x* is the ‘‘minimum’’ alignment which could contain it) in a manner analogous to the normal padding operations for offsets. This offset could then used both in the construction of a compound shape and in the extraction and assignment constructors.

The assignment of bitfields follows the same pattern with the same constraints using **bitfield_assign**:

arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
arg3:	EXP BITFIELD_VARIETY(*v*)
	→ EXP TOP

Chapter 8

Operations

Most of the arithmetic operations of XANDF have familiar analogues in standard languages and processors. They differ principally in how error conditions (for example, numeric overflow) are handled. There is a wide diversity in error handling in both languages and processors, so XANDF 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 as to 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, XANDF 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, 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 XANDF 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 either to ignore it, to allow one to jump to a label, or to obey an XANDF exception handler if it happens.

8.1.1 ERROR_TREATMENT

Taking integer addition as an example, **plus** 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 *ov_err*.

The ERROR_TREATMENT **impossible** is an assertion by the producer that overflow will not occur, otherwise the action is undefined.

The ERROR_TREATMENTS **continue** and **wrap** give ‘‘fixup’’ values for the result. For **continue** the fixup value is undefined. For **wrap**, the the answer will be modulo 2 to the power of the number of bits in the representational variety. Thus, integer arithmetic with byte representational variety is done in modulo 256. This just corresponds to what happens in most processors and, incidentally, also to the definition of C.

The ERROR_TREATMENT that one would use if one wished to jump to a label is **error_jump**:

lab:	LABEL
	→ ERROR_TREATMENT

A branch to *lab* will occur if the result overflows.

The ERROR_TREATMENT, **trap***(overflow)* will raise an XANDF exception (see Section 6.3.0) with ERROR_CODE **overflow** if overflow occurs.

8.2 Division and Remainder

The various constructors in involving integer division (for example, **div1**, **rem1**) have two ERROR_TREATMENT parameters, one for overflow and one for divide-by-zero. For example, **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** and **rem2** 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 has 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 constructors **div0** and **rem0** allow the translator to choose whichever of the two forms of division is convenient — the producer is saying that he does not care which is used, as long as they are pair-wise consistent. The precise definition of the divide operations is given in the section ‘‘Division and Modulus’’ in the referenced **XANDF_PS** specification.

8.3 change_variety

Conversions between the various INTEGER varieties are provided for by **change_variety**:

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**, **not**, **or** and **xor** are provided for all integer varieties. Since integer varieties are defined to be represented in twos-complement, the results of these operations are well defined.

8.5 Floating-point Operations, ROUNDING_MODE

All of the floating-point (including complex) 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** and **floating_mult** 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 performed by **float_int**, and from floating to integers by **round_with_mode**. This latter constructor has a parameter of SORT ROUNDING_MODE which effectively gives the IEEE rounding mode to be applied to the float to produce its integer result.

One can extract the real and imaginary parts of a complex FLOATING using **real_part** and **imaginary_part**. A complex FLOATING can be constructed using **make_complex.** Normal complex arithmetic applies to all the other FLOATING constructors except for those explicitly excluded (for example, **floating_abs**, **floating_max**, and so on).

8.6 change_bitfield_to_int, change_int_to_bitfield

There are two bit-field operations, **change_bitfield_to_int** and **change_int_to_bitfield**, to transform between bit-fields and integers. If the varieties do not fit, the result is undefined; the producer is assumed to prevent this situation.

8.7 make_compound, make_nof, n_copies

There is one operation to make values of COMPOUND SHAPE **make_compound**:

arg1: EXP OFFSET(*base,y*)
arg2: LIST(EXP)
→ 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**, **make_nof_int** and **n_copies**. Again, they only occur in C as constants in global definitions.

Chapter 9

Constants

The representation of constants clearly has special difficulties in any architecture neutral format. Leaving aside any problems of how numbers are to be represented, there is also 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 XANDF, 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. XANDF assumes that, if these EXPs can be evaluated at translate-time (that is, 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[6]; 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; this obviously includes any **make_proc** or **make_general_proc**. 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 XANDF translator effectively has an XANDF interpreter which can do evaluation of expressions (including conditionals, and so on) 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 the section on ‘‘Constant Evaluation’’ in the referenced **XANDF_PS** specification.

9.1 _cond Constructors

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, for example, **exp_cond**:

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

[6] However, see also **initial_value** in Section 3.3 on page 14.

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 XANDF. This kind of construction is used extensively in C pre-processing directives, for example:

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

9.2 Primitive Constant Constructors

Integer constants are constructed using **make_int**:

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*.

Note: In this document, where the text might read ‘‘1 as an integer EXP’’ (for example), a more accurate script would read ‘‘**make_int**(*v*, 1) where *v* is some appropriate VARIETY.’’

Constants for both floats and strings use STRINGs. A constant string is just an particular example of **make_nof_int**:

v:	VARIETY
str:	STRING(*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 ASCII-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. The elements of a Unicode string would be integers of size 16 bits.

A floating constant uses a STRING which contains the ASCII characters of a expansion of the number to some base in **make_floating**:

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

For a normal floating point number, each integer in *mantissa* is either the ASCII ‘‘.’’ symbol or the ASCII representation of a digit of the representation in the given base; that is, if c is the ASCII symbol, the digit value is the ASCII value of *c* minus the ASCII value of *‘‘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.

The **make_floating** construct does not apply apply to a complex FLOATING_VARIETY *f*; to construct a complex constant use **make_complex** with two **make_floating** arguments.

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

Chapter 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, for example, the EXP OFFSETS for fields of structures. However, the real justification for TOKENs is 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. An XANDF 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 XANDF abstracts FILE as a SHAPE TOKEN with no parameters, for example, **make_tok** (T_FILE). Any program that uses FILE would have to include a TOKDEC introducing T_FILE:

```
make_tokdec(T_FILE, empty, 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 an XANDF 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 XANDF 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. The actual API expansion for the **putc** value is rarely anything 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 XANDF linking and system linking for that platform. This is illustrated in Figure 10-1, which gives the various phases involved in producing a program executable.

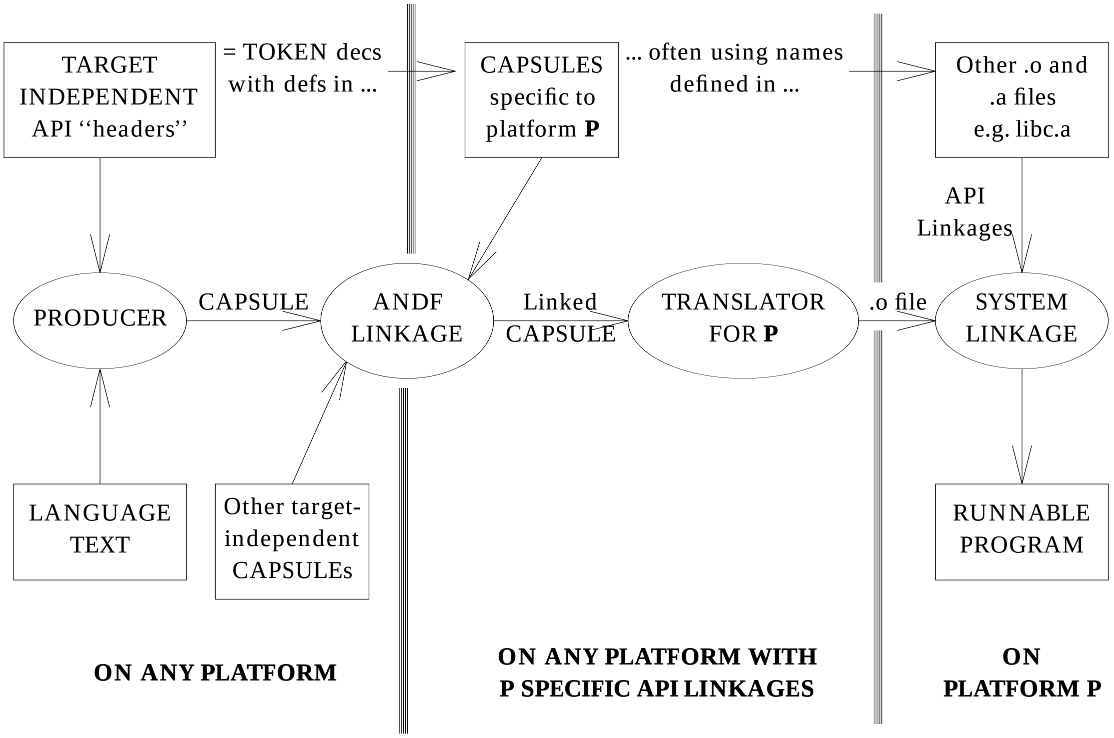

TARGET INDEPENDENT API ‘‘headers’’
= TOKEN decs with defs in ...
CAPSULES specific to platform **P**
... often using names defined in ...
Other .o and .a files e.g. libc.a
API Linkages
PRODUCER
CAPSULE
ANDF LINKAGE
Linked CAPSULE
TRANSLATOR FOR **P**
.o file
SYSTEM LINKAGE
LANGUAGE TEXT
Other target-independent CAPSULEs
RUNNABLE PROGRAM
ON ANY PLATFORM
ON ANY PLATFORM WITH P SPECIFIC API LINKAGES
ON PLATFORM P

Figure 10-1 XANDF 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 an XANDF 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**, and so on, 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**. For a world-wide standard API, the EXTERNAL name for a TOKEN used by **make_linkextern** should be provided by an application of **unique_extern** 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**.

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 re-order 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 XANDF 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, the viewpoint has been as though a TOKENised API could only be some library interface, built on top of some language, like XPG3, POSIX, X, and so on, 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. This TOKEN could be expanded in XANDF 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**, and so on.

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 XANDF 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.

In addition, LPIs would be a very useful tool for crystallising 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 XANDF.

Chapter 11

XANDF Transformations

XANDF to XANDF transformations form the basis of most of the tools of XANDF, including translators. XANDF has a rich set of easily performed transformations, 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))
```

that is, 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 XANDF constructions so that the code-production phase of the translator can just proceed 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, or 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, XANDF transformations can be used as the basis for defining XANDF. In principle, if we were to define all of the allowable transformations of the XANDF constructions, we would have a complete definition of XANDF as the initial model of the XANDF 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 XANDF 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 difficult 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[7] EXP constructor and E is one of the EXP operands of F, then:

```
F(... , E, ...)
→ identify(empty, newtag, E, F(... ,obtain_tag(newtag), ...))
```

{B} If E is a non side-effecting[8] EXP and none of the variables used in E are assigned to in B:

```
identify(v, tag, E, B) → 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)
→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)
↔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)
↔sequence(($S_1$, ..., $S_{i-1}$, $P_1$, ..., $P_m$, R, $S_{i+1}$, ..., $S_n$), T)
```

{F}

```
E ↔ sequence(( ), E)
```

{G} If D is either identify or variable:

```
D(v, tag,sequence(($S_1$, ..., $S_n$), R), B)
→ 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)
→ 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) → E
```

[7] The order-specifying constructors are **conditional**, **identify**, **repeat**, **labelled**, **sequence** and **variable**.

[8] 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.

{J} If S_i is make_top(), then:

sequence((S_1, S_2, ... S_n), R)

↔ 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())

↔ sequence((S_1, ..., S_{n-1}), S_n)

{L} If E is an EXP TOP and E is not side-effecting then

E → 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 $P_{1 \ldots m}$ are the EXP operands of C:

sequence((S_1, ..., S_n), R)

→ 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)

→ sequence((S_1, ..., S_n),conditional(L, R, A))

{O} If there are no uses of L in X[9]:

conditional(L, X, Y) → X

{P}

conditional(L, E ,goto(Z)) → E[L\Z]

{Q} If EXP X contains no use of the LABEL L:

conditional(L,conditional(M, X, Y), Z)

→ conditional(M, X,conditional(L, Y, Z))

{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())))

[9] There are analogous rules for **labelled** and **repeat** with unused LABELs.

{U} If f is a procedure defined[10] 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, (A1, ..., An), V)
→variable(empty, newtag, make_value((rshape=BOTTOM)? TOP: rshape),
     labelled( (L),
         variable(v1, tg1, A1, ..., variable(vn, tgn, An,
             variable(empty, vtg, V,
                 B(sequence(assign(obtain_tag(newtag), R1),goto(L)), ...,
                     sequence(assign(obtain_tag(newtag), Rm),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, ((E1, D1), ..., (En, Dn)))
    → variable(empty, nt,make_value(COMPOUND(S) ),
      sequence(
          (assign(add_to_ptr(obtain_tag(nt), D1), E1),
              ...,
              assign(add_to_ptr(obtain_tag(nt), Dn), En)),
          contents(S,obtain_tag(nt) )
      )
    )
```

[10] This has to be modified if B contains any uses of **local_free_all** or **last_local**.

11.1.1 Examples of Transformations

Any of these transformations may be performed by the XANDF translators. The most important is probably {A} which allows one to reduce all of the EXP operands of suitable constructors to **obtain_tags**. The expansion rules for identification, {G}, {H} and {I}, gives definition to complicated operands as well as strangely formed ones, for example, **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 inlining given by {U} or loop unrolling 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()**). Inlining 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_tags** can be eliminated by rule {M} and then the sequences by {F}. Successive 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 XANDF 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.[11] Any value which is dependent on the effect or result of an undefined construction is also undefined. This is not the same as 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.

[11] However, the mapping of a constraint may allow 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 Chapter 12).

Chapter 12

XANDF 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:[12]
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 up to 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.

[12] shape_offset(sh_int) would be equally valid and simpler for S_i. However the formation above is more uniform with respect to selection OFFSETs.

From the definition, one finds that:

```
S_c = shape_offset(SH_mystruct)
```

that is, an:

```
OFFSET(alignment(sh_int) ∪alignment(sh_char) ∪alignment(sh_double), {})
```

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)
→ EXP POINTER(x)

... 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 XANDF, perhaps by re-ordering the fields, padding the offsets by later alignments or taking maxima of the sizes of the fields.

12.1 Bitfield Offsets

Bitfield offsets are governed by rather stricter rules. In order to extract or assign a bitfield, one has to find an integer variety which entirely contains the bitfield. Suppose that one wished to extract a bitfield by:

```
bitfield_contents(v, p:POINTER(X), b:OFFSET(Y, B))
```

Y must be an **alignment** (I) where I is some integer SHAPE, contained in X. Further, this has to be equivalent to:

```
bitfield_contents(v,add_ptr(p, d:OFFSET(Y,Y)), b′ :OFFSET(Y, B))
```

for some d and b’ such that:

```
offset_pad(v,shape_offset(I) ) >= b′
```

and:

```
offset_add(offset_pad(v,offset_mult(d, sizeof(I)), b′) = b
```

Clearly, there is a limitation on the length of bitfields to the maximum integer variety available. In addition, one cannot have a bitfield which overlaps two such varieties.

The difficulties inherent in this may be illustrated by attempting to construct an array of bitfields using the **nof** constructor. Assuming a standard architecture, one cannot usefully define an object of SHAPE **nof** (N, **bitfield**(**bfvar_bits**(b, M))) without padding this shape out to some integer variety which can contain M bits. In addition, they can only be usefully indexed (using **bitfield_contents**) either if M is some power of 2 or M*N is less than the length of the maximum integer variety. Thus a producer must be sure that these conditions hold if they are to generate and index this object simply. Even here, the producer is in some difficulty, since they do not know the representational varieties of the integers available to them; also it is difficult for them to ensure that the alignment of the entire array is in some sense minimal. Similar difficulties occur with bitfields in structures — they are not restricted to arrays.

The solution to this conundrum in its full generality requires knowledge of the available representational varieties. Particular languages have restrictions which mean that sub-optimal solutions will satisfy its specification on the use of bitfields. For example, C is satisfied with bitfields of maximum length 32 and simple alignment constraints. However, for the general optimal solution, there is no reasonable alternative to the installer defining some tokens to produce bitfield offsets which are guaranteed to obey the alignment rules and also give optimal packing of fields and alignments of the total object for the platform in question. Three tokens are sufficient to do this. These are analogous to the constructors **offset_zero**, **offset_pad** and **offset_mult** with ordinary alignments, and their signatures could be:

Bitfield_offset_zero:

n:	NAT
issigned:	BOOL
	→ EXP OFFSET(A, *bfvar_bits(issigned,n)*)

Here the result is a zero offset to the bitfield with ‘‘minimum’’ integer variety alignment A.

Bitfield_offset_pad:

n:	NAT
issigned:	BOOL
sh:	SHAPE
	→ EXP OFFSET(*alignment(sh)* ∪ A, *bfvar_bits(issigned,n)*)

Here the result is the **shape_offset** of *sh* padded with the ‘‘minimum’’ alignment A so that it can accommodate the bitfield. Note that this may involve padding *sh* with the alignment of the maximum integer variety if there are not enough bits left at the end of sh.

Bitfield_offset_mult:

n:	NAT
issigned:	BOOL
ind:	EXP INTEGER(*v*)
	→ EXP OFFSET(A, *bfvar_bits(issigned,n)*)

Here the result is an offset which gives the displacement of ind^{th} element of an array of *n*-bit bitfields with ‘‘minimum’’ alignment A. Note that this will correspond to a normal multiplication only if *n* is a power of 2 or $ind*n \leq$ length of the maximum integer variety.

These tokens can be expressed in XANDF if the lengths of the available varieties are known, that is, they are installer defined.[13] They ought to be used in place of **offset_zero**, **offset_pad** and

offset_mult wherever the alignment or shape (required to construct a SHAPE or an argument to the bitfield constructs) is a pure bitfield. The constructor **nof** should never be used on a pure bitfield; instead it should be replaced by:

```
S = compound( Bitfield_offset_mult(M,  b, N))
```

to give a shape, S, representing an array of N M-bit bitfields. This may not be just N*M bits. For example, **Bitfield_offset_mult** may be implemented to pack an array of 3-bit bitfields as 10 fields to a 32-bit word. In any case, one would expect that normal rules for offset arithmetic are preserved, for example:

```
offset_add( Bitfield_offset_pad(M,b,S),size(bitfield(bfvar_bits(b,N) )))
  =  Bitfield_offset_mult(M,  b, N+1)
```

where:

```
size(X) = offset_pad(alignment(X),shape_offset(X) )
```

[13] For most architectures, these definitions are dependent only on a few constants such as the minimum length of bitfield, expressed as tokens for the target. The precise specification of such target dependent tokens is of current interest outside the scope of this document.

Chapter 13

Models of the XANDF Algebra

XANDF is a multi-sorted abstract algebra. Any implementation of XANDF 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 XANDF is independent of context: it is a homomorphism. In other words if we define the mapping of an XANDF constructor, C, as MAP[C] and the representation of a SORT, S, as REPR[S], then:

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

Any mapping has to preserve the equivalences of the abstract algebra, such as those exemplified by the transformations {A} - {Z} described in Section 11.1 on page 60. 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 an XANDF 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 XANDF 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, XANDF 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:

```
REPR[A ∪ B] = Max(REPR[A], REPR[B])
```

that is:

```
MAP[unite_alignment] = Max
```

while the ALIGNMENT inclusion predicate is given by:

```
REPR[A ⊃ B] = REPR[A] ≥ 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 ∪ and ⊃ must preserve the basic algebraic properties derived from sets. For example, the mapping of ∪ 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)]≠1 is represented by a 32-bit byte address.

It is not allowed to construct pointers where REPR[A]=1, but there are still offsets where the second alignment is a bitfield. Thus offsets to bitfield are represented differently to offsets to other alignments:

- A value with SHAPE OFFSET(A, B) where REPR(B)≠1 is represented by a 32-bit byte-offset.
- A value with SHAPE OFFSET(A, B) where REPR(B)=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(top() )] = 0
REPR[shape_offset(integer(char_variety) )] = 8
REPR[shape_offset(integer(short_variety) )] = 16
```

.... and so on, for other numeric varieties.

```
REPR[shape_offset(pointer(A) )] = 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) )]
```

where S is not a bitfield shape.

Similar considerations apply to the other offset-arithmetic constructors. In general, we have:

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

REPR[offset_pad(A, X:OFFSET(C,D))
    = ((REPR[X] + REPR[A]-1)/(REPR[A]))*REPR[A]/8
    if REPR[A] ≠ 1 Λ REPR[D ] = 1
Otherwise:
REPR[offset_pad(A, X:OFFSET(C,D))
    = ((REPR[X] + REPR[A]-1)/(REPR[A]))*REPR[A]

REPR[offset_add(X:OFFSET(A,B), Y:OFFSET(C,D))]
    = REPR[X] *8 + REPR[Y]
    if REPR[B] ≠ 1 Λ REPR[D] = 1
Otherwise:
REPR[offset_add(X,Y)] = REPR[X] + REPR[Y]

REPR[offset_max(X:OFFSET(A,B), Y:OFFSET(C,D))]
    = Max(REPR[X], 8*REPR[Y]
    if REPR[B] = 1 Λ REPR[D] ≠ 1
REPR[offset_max(X:OFFSET(A,B), Y:OFFSET(C,D))]
    = Max(8*REPR[X], REPR[Y]
    if REPR[D] = 1 Λ REPR[B] ≠ 1
Otherwise:
REPR[offset_max(X,Y)] = Max(REPR[X], REPR[Y])

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

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}] ≥ 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 such as integers, floats, displacements, and so on. There are at least two different kind of blocks, one which can only contain block-addresses and the other which contains only scalar data. Clearly, there are 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)
```

... and so on, 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∪B] = (s:Max(REPR[A].s, REPR[B].s), b:REPR[A].b ∨ REPR[B].b)
```

and their inclusion relationship is given by:

```
REPR[A⊃B] = (REPR[A].s ≥ REPR[B].s) ∧ (REPR[A].b ∨ ¬ REPR[B].b)
```

13.2.2 Offset and Pointer Model

A value with SHAPE POINTER A where ‘‘¬ 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 ‘‘¬ 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
```

... and so on, 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 Λ ¬ 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 Λ 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 Λ ¬ REPR[C].b

REPR[offset_pad(A, Y: OFFSET(C,D) )]
    = (REPR[Y] + REPR[A].s - 1)/REPR[A].s
    if ¬ REPR[A].b Λ ¬ 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 Λ ¬ REPR[C].b
REPR[offset_max(X: OFFSET(A,B), Y: OFFSET(C,D) )]
    = Max(REPR[X], REPR[Y])
    if ¬ REPR[A].b Λ ¬ 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 Λ 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 Λ ¬ 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 Λ ¬ REPR[A].b

REPR[offset_subtract(X: OFFSET(A,B), Y: OFFSET(C,D) )]
    = REPR[X] - REPR[Y]
    if ¬ REPR[A].b Λ ¬ 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 Λ REPR[C].b
REPR[offset_add(X:  OFFSET(A,B), Y: OFFSET(C,D))]
    = REPR[X].sd - REPR[Y]
    if REPR[A].b Λ ¬ 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.

Glossary

$\cap$
: cap (intersection)

$\cup$
: cup (union)

$\subset$
: subset of

$\supset$
: superset of

$\subseteq$
: improper subset of

$\supseteq$
: improper superset of

ABI
: Application Binary Interface

ANDF
: Architecture Neutral Distribution Format

ANSI
: American National Standards Institute

API
: Application Programming Interface.

argument
: Information which is passed to a function or operation and which specifies the details of the processing to be performed.

atom
: An atom is a unique ID corresponding to a string name. Atoms are used to identify properties, types and selections.

compiling
: production of ANDF from some source language.

DRA
: The United Kingdom Defence Research Agency.

homomorphism
: many-to-one mapping which retains the structure

installing
: linking and mapping of ANDF onto a concrete machine using processor-specific libraries implementing the API calls and data formats.

LPI
: Language Programming Interface

POSIX
: IEEE Portable Operating System Interface

producing

production of ANDF from some source language (same meaning as ‘‘compiling’’).

translating

making a program for some specific platform from ANDF

XANDF

X/Open specification for the Architecture Neutral Distribution Format

Index