Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Rules for Type Inferencing

Nicola Howarth

Abstract

This document explores the methods by which type inferencing can be carried out in a type system based on type conformance for subtyping, such as the ODP and ANSA computational languages.

APM.1552.01 **Approved** 23rd August 1995

Technical Report

Distribution:
Supersedes:
Superseded by:

Rules for Type Inferencing

Rules for Type Inferencing

Nicola Howarth

APM.1552.01

23rd August 1995

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This document explores the methods by which type inferencing can be carried out in a type system based on type conformance for subtyping, such as the ODP and ANSA computational languages. The work was conducted using the ANSA prototype language DPL as an exemplar [ANSA 93]. Some background to the work is given in Chapter 2. This draws on previous work, which is described more fully in [Watson 92]. The syntax of DPL is given in Chapter 3, and the inference rules for each syntactic element are described in Chapter 4.

An implementation of the rules discussed here is being carried out on an implementation of a C++ DPL which makes use of abstract syntax trees. This implementation is described in [Howarth 94].

2 Background

2.1 A brief history of DPL

DPL was originally documented in Volume C of the ANSA Reference Manual [APM89]. This description included a syntax for the language along with type inference rules for each of the syntactic expressions. It was used as a vehicle to experiment with concepts for the ANSA and ODP computational models.

The initial experimental implementation of DPL was provided by a compiler written in and generating Common Lisp, running on a Xerox Lisp machine. The syntax had evolved from that described in the ARM, but the underlying computational model was the same, and so the semantics of the language differed little. This implementation was subsequently made available via an electronic mail service with the name “dpltools”, and will be referred to as “DPLtools” throughout this document.

DPLtools was also the prototype implementation of a type checker for DPL. Although the intention was to implement early type checking, it was not immediately apparent how to do this in all cases, so an “early checking where possible” philosophy was adopted. The dynamic type system of the underlying Lisp implementation served to catch type errors which the DPL type checker could not.

Although DPLtools is still available, having survived the retirement of the Xerox lisp machines by being translated into an Emacs Lisp package running within the GNU Emacs editor, implementation effort has now switched to providing a DPL-to-C translator which can serve as successor to ANSAware’s PREPC and IDL preprocessors, allowing C programs to be written that use the facilities of the ANSAware distributed system infrastructure by way of DPL annotations. Concurrently with this implementation effort, the present study was undertaken to resolve the type system issues which had become better understood through experimenting with DPLtools.

DPLtools’ type system was largely based on that of Emerald [Hutchinson87], a language for distributed programming with which it has many goals in common. The detailed differences between their two type systems stem largely from differences between the models behind the two languages, although a couple of features were also omitted out of a desire to find a better way of achieving their effect, without quite knowing how.

This chapter summarises the salient features of DPLtools’ type system. It assumes familiarity with the ANSA computational model [APM91a], and an acquaintance with DPL, as described in the ANSA Programmers’ Manual [Howarth91]. Current versions of that document are deliberately vague on the details of types in DPL, since the type system implemented at the time was known to have shortcomings. This chapter provides the description of the DPLtools type system left out of the Programmers’ Manual.

2.2 Summary of the ANSA computational model

A brief summary of the computational model follows for those without access to the full description.

The ANSA computational model specifies a set of abstractions that can be used for constructing distributed applications. It is based on *objects* whose encapsulated state is manipulated via *services* made available at one or more *interfaces* presented by the object. The primitives of the service are its *operations*, each of which may require some number of argument parameters and generate one of a set of named outcomes (called *terminations*) carrying one or more result parameters.

Invocation of an operation in an interface requires the caller to possess an *interface reference* for that interface instance. Invocation is synchronous (that is, the caller blocks while the operation executes and continues once it has terminated); this is modelled as the calling *activity* (thread of control) migrating from the caller to the called operation and then back again. Activities running code within an object may create new interfaces to that object by executing *an interface constructor* when required; this causes the new interface to be created and returns the only *interface reference* to it as a result. An interface reference may be freely duplicated by its holder. Within the computational model all operation arguments and termination parameters are references to services.

An activity may also create a new object by executing an *object constructor*; the object thus created has no special relationship to the object which contains the constructor, other than possibly being given references to interfaces provided by the newly-created object.

Objects, interfaces and their associated state continue to exist as long as they are accessible via interface references.

A distributed program constructed using the ANSA computational model is executed by first transforming it into an equivalent engineering description tailored to the particular platform technology and configuration required. The details of the engineering model are not especially relevant to this discussion: the important point is that the transformation is entirely automated by tools which specialise the general computational description into an engineering description tailored to a particular configuration.

All interactions in systems built using the ANSA computational model take place at interfaces, and hence within the model it is the interfaces that are typed. The computational model type system is built on *type conformance*, which can be defined in terms of substitutability; one type is said to conform to another if an interface of the first type can be substituted for one of the second type without any possible client ever experiencing an *interaction error* (“method not understood”).

The ODP Computational Model (ISO 10746-3/ITU X.903) defines a similar type system [ODP95].

2.3 Type conformance

Type conformance is fully described in section 2.3 of [AR.001]. In this paper the notation a $\Rightarrow$ b will be used to mean “a conforms to b”. It will also be useful to define a couple of terms for use in describing conformance-based type

systems. The terms *wider* and *widen*, and correspondingly *narrower* and *narrow*, will sometimes be used:

Narrower

This term is used somewhat informally to describe a type that has fewer operations than another, or has the same operations with narrower result parameters or wider argument parameters.

Wider

Also used informally to describe a type that has more operations than another, or has the same operations with wider result parameters or narrower argument parameters.

Note that describing one type as wider or narrower than another is not intended to signify that there is any conformance relationship between the two.

For any pair of types we will often need to discuss the types that conform to both, or to which both conform. The terms *meet* and *join* are useful here:

Meet

For any pair of types X, Y there is a non-empty set of types to which both X and Y conform. The meet of X and Y is the member of this set that conforms to all other members of the set (informally, it's the widest type to which both X and Y conform).

Join

For any pair of types X, Y there is a set of types that conform to both X and Y (although in the DPL type system this set may be empty). The join of X and Y is the member of this set to which all other members conform (informally, it's the narrowest type that conforms to both X and Y).

Note: These definitions of meet and join assume that there is precisely one type in each set to which all meets and joins conform in the appropriate direction. We have not actually established (i) if this is always true and (ii) if it isn't, whether it matters.

The relationship between a specific pair of types and their meet and join is illustrated diagrammatically in figure 2.1.

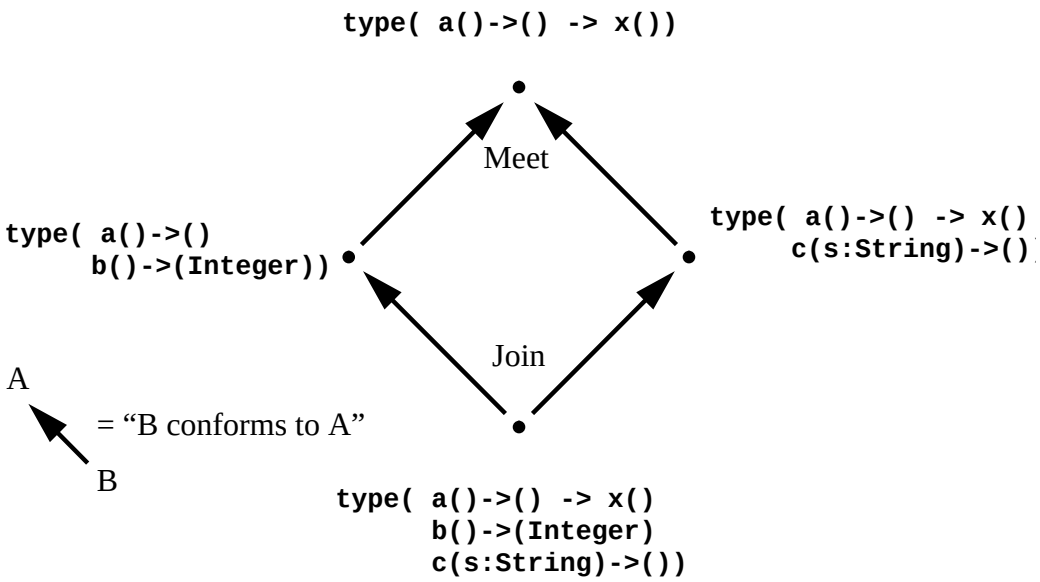
It may be a useful aide-memoire to think of the join of the two types as having the arrows representing conformance joined together at the base, while the meet lies where the conformance arrowheads meet.

2.4 Manifest and latent types

DPL was designed as a realisation of the construction model of the ANSA computational model. Every computational model interface has a conceptual *latent type*; the set of named operations that the interface supports along with the parameter types to which the actual (latent) parameter types must conform if there is not to be an interaction error.

In order to support early type checking it is necessary to assign a *manifest type* to each syntactic expression in a program such that the latent types of all the interfaces that can be produced by evaluating the expression can safely be manipulated as if they had that manifest type, using some suitably-defined *substitutability relationship*.

Figure 2.1: Meet and Join



In the type system used by DPLtools both the latent and manifest types were full computational model types, and the substitutability relationship between them was type conformance. For any expression there are in general many possible manifest types that would satisfy the conformance invariant, but the most useful one is the widest, since that maintains the most information about the operations to which the service concerned will respond.

2.5 Type inference

In order to ease the task of the programmer and reduce the scope for error, DPL was designed so that the manifest types of as many expressions as possible could be inferred from the program text without the use of explicit type declarations. Explicit declarations of types (using *type expressions* written in syntactic *type positions*) are only required in the following expressions:

- When declaring the signatures of operations, a manifest type has to be specified for the formal argument and result parameters. In a type-correct program the manifest types of the expressions used as actual parameters have to conform to these formal parameter types.
- When handling terminations, manifest types for the formal parameters to the handler have to be given, and for type correctness the manifest types of the expressions generating the actual parameters have to conform to these formal parameter types.
- The manifest types of variable bindings have to be declared; expressions generating values to be bound to those names must have manifest types that conform to that type, and any reference to the variable binding is considered to have that type. The type of a variable binding must be declared because the type checker cannot predict the manifest types of all the values that the programmer may wish to store in the binding over its life.

The manifest types of all other expressions in the language are inferred from the manifest types of their constituent expressions using inference rules that attempt to derive the widest possible manifest type (subject to the conformance invariant). The algorithms used are derived from those documented in part X section 7.4 of The ANSA Reference Manual [APM89], which applied to an earlier version of DPL.

The starting point for type inference is the `interface(...)` constructor, since all interfaces in a DPL program are originally created in this way[1]. By virtue of the fact that the text of the constructor completely specifies the interface that it creates, the manifest type of this expression is the same as the latent type of the interface produced. However, the use of the conformance relation to permit interfaces of a wider type to be used where narrower ones are required, and the inability of the type inferencer to predict the flow of control in the program at run time soon conspire to produce a divergence between the manifest type of an expression and the latent types of the interfaces that may be produced by executing that expression. To illustrate this, consider the example in figure 2.2.

Figure 2.2: Latent vs Manifest types

```
x = interface( a()->() [ {body} ]
               print(s:OutputStream)->() [ {body} ]
             )

y = interface( b()->() [ {body} ]
               print(s:OutputStream)->() [ {body} ]
             )

z = after  ( {boolean test} ) % Inferred manifest type of z is
the
    handle ( true() [x]       % meet of types of x and y
             false  [y]
           )
z.print(StandardOutput);      % Never causes interaction error
z.a();                        % May cause interaction error
```

In this example the manifest type of the constant binding `x` is the same as the latent type of the interface in the binding (and will always be so, since the interface bound to the name cannot be changed), and similarly the manifest type of `y` is the same as the latent type of the item in `y`. However, depending on the outcome of the run-time evaluation of the boolean test, the latent type of the interface bound to `z` could be either of the first two. Therefore, in order for the program to be type-safe, `z` must be assigned a manifest type to which the manifest types of both `x` and `y` conform. The widest such type is the meet of the types of `x` and `y`, in this case the type that has only the `print` operation.

It can therefore be seen that early type checking involves examining the program text to make sure that the results of expressions are only ever manipulated in accordance with their manifest types. If all type checking and analysis is performed at compile-time then the latent types of interfaces need not be explicitly represented, but remain implicit in the set of interactions in which the interface can take part; the manifest type information used (and

1. This ignores the issue of what the `type(...)` constructor creates, for the moment.

then discarded) at compile-time is sufficient to guarantee that no interaction with an interface will produce an interaction error.

2.6 Polymorphism

The use of a substitutability rather than an equivalence relation between the latent types of interfaces and the manifest types of the argument positions in which they can be passed to operations provides what Cardelli and Wegner [Cardelli85] term *inclusion polymorphism*. This permits interfaces with many different latent types to be viewed as having the same manifest type and therefore manipulated in the same way by the same operation body.

Since the body of an operation is (in general) not available to be inspected by the compiler generating calls to that operation, the operation type must include a specification of the way that the body will manipulate the parameters, so that the caller's compiler can ensure that the caller will only pass parameters that can legally be manipulated in that way. In DPLtools this is done by declaring in the operation signature the manifest type that each parameter will be considered to have within the operation body. Since the conformance relation is transitive, and the type checker maintains the invariant that the latent type of the actual argument will conform to the manifest type of the expression that generates it, it is safe to invoke the operation on the result of any expression whose manifest type conforms to the formal parameter type.

This ability to specify the instances to which an operation can be applied by giving a type to which all of their types must conform is termed *bounded quantification* by Cardelli and Wegner.

2.7 Document exclusions

The type representation system implemented is capable of representing, and checking the substitutability of interface types consisting of named operations, each of which can return one of a number of named terminations. Both operations and terminations can carry one or more parameters, each of which themselves has a known, fixed interface type.

Because the precise types of all paramters and results are known in advance, the substitutability of types in this system can be determined statically, independently of how the program executes, and all possible type errors can be identified at compile-time or link-time. However, by the same token, this type system is not capable of accurately representing the type of any service where the type returned by an operation is not a constant.

While this is not in general a restriction, there are two important classes of service whose types cannot be accurately represented in a static type system, precisely because of this lack of type paramterisation:

- The interfaces presented by a trader, or any other service which serves to maintain and return information about heterogenous collections of services.
- The interfaces presented by factories for homogeneous collections, and other services where the type of a result is dependent on a type parameter.

The first case is fairly straightforward - we need to express the types of operations where the type of a result (or a parameter) is the same as a type

passed in as a parameter. Provided we arrange that the type used to parameterise the operation is always known at type-checking time then these parameterised operations can also be considered to have a fixed (static) type, notwithstanding that their types are unknown before the parameterisation. This approach to precisely expressing the types of a trader's import and export operations is discussed and analysed in detail in [ABAD89] and [ABAD91], in the precisely analogous context of importing and exporting concrete data structures to files. Although our type system cannot currently represent this form of unbounded type parameterisation, the underlying theory seems well understood, so creating syntax to represent such types, altering the data structures to represent them and the type checker to determine conformance should be a Small Matter Of Programming.

The second case, that of expressing factory types, and in particular of factories for homogeneous collections, is more problematic. The implementation of many collection types requires the code implementing the collection to perform operations on the member elements; for instance, a set cannot by definition contain any duplicates, so the code that implements the set must be able to call an equal operation on pairs of elements in the set to discover possible duplication. In order to build a generic factory for sets that can be parameterised by a type in order to produce a set of a specific type, one must therefore impose (and express) a constraint on the type used as the parameter - in this case something like “must posess an equal operation that takes another instance of the same type as a parameter”. It turns out that this form of bounded type parameterisation is very difficult to express. Although some research has been done along some hopeful avenues (such as F-bounded quantification [CANN89] [KATI92]), no complete theory for such a system has been proved to be type-safe. However, it would be possible to modify the type syntax and implementation to support some plausible guesses on type-parameter bounding, as has been done in the past, though without any guarantee of correctness, and bearing in mind that there are plausible-looking type systems which have in the past eventually been proved to be undecidable [PIER92].

3 Syntax Rules

activity → **activity** *block*

argumentList → **(** { *declaration* } **)**

assignment → *name* { *name* } **:=** *expression*

attributeList → **<** { *attributeName* [*attributeBlock*] } **>**

block → **(** [*expressionList*] **)** | **[** [*expressionList*] **]**

declaration → *name* { *name* } **:** *typeExpression*

defaultHandler → **?** *block*

definition → *name* { *name* } **=** *expression*

expression → *activity* | *assignment* | *definition* | *handledBlock* | *initialisation* | *interface* | *literal* | *terminate* | *typeDefinition* | *unit*

expressionList → *expression* [{ **;** *expression* } | { **,** *expression* } | { **||** *expression* } | { **|** *expression* }]

handledBlock → **after** *block* **handle** **(** { *namedHandler* } [*defaultHandler*] **)**

initialisation → *declaration* { *declaration* } **:=** *expression*

interface → **interface** [*attributeList*] [**data** [*language*] *embedded*] **(** { *signature operationBody* } **)**

invocation → *unit* **.** *operationName* *block*

namedHandler → *terminationName* *argumentList* *block*

object → **object** [*attributeList*] [**data** [*language*] *embedded*] *block*

operationBody → *block* | **code** [*language*] *embedded*

program → *object*

resultList → **(** { *typeExpression* } **)**

signature → *operationName* [*attributeList*] *argumentList* **->** [*terminationName*] *resultList* { **->** *terminationName* *resultList* }

terminate → **->** *terminationName* *block* | **-> reterminate**

typeBlock → **(** *typeExprList* **)**

typeConstructor → **type** [*attributeList*] **(** { *signature* } **)**

typeDefinition → *typeName* { *typeName* } **=** *typeExpression*

typeExpression → *typeName* | *typeConstructor* | *typeBlock*

typeExprList → *typeExpression* [{ **,** *typeExpression* }]

unit → *name* | *invocation* | *block* | *object*

4 Inference Rules

4.1 “Types of Types”

The following glossary provides a helpful reference:

- Interface type: a collection of operation types, each labelled by an operation name.
- Operation type: has a parameter list (an interface list type and list of corresponding positional parameter names) and a a response type.
- Response type: a collection of termination types, each labelled by a termination name. There may be an anonymous termination (i.e. labelled by a null name).
- Termination type: an interface list type.
- Interface List type: an ordered list of interface types.

4.2 Binding Expressionss

assignment → *name* | { *name* | } **:=** *expression*

initialisation → *declaration* { *declaration* } **:=** *expression*

definition → *name* { *name* } **=** *expression*

There are three forms of binding expression: *assignment* and *initialisation*, which are variable binding expressions, and *definition*, which is a fixed binding expression.

4.2.1 Variable bindings - initialisation and assignment

A list of the types of the names/declarations must be determined. In the case of initialisation, this will also involve setting up the type for each name in the symbol table. For assignment, the type will be retrieved from the symbol table. The types of the expressions are also determined. The anonymous termination on the right is then checked for conformance to the list on the left. Possible errors are lack of conformance, and no anonymous termination. The type of the initialisation is the type of the right hand side.

4.2.2 Fixed bindings

For a constant binding, the types on the names are set to empty (see below), and the type of the expression is determined. The types of the bindings are then set according to the anonymous termination of the expression. An error occurs if there is no anonymous termination on the right-hand side, or if the list is the wrong length.

In order to handle typing of recursive definitions we need to set the types of the LHS names, find the type of the expression in that environment, then go

back and correct the LHS (thus setting up the circularity, if required). Any initial type will do.

4.3 Block

block → **(** [*expressionList*] **)** | **[** [*expressionList*] **]**

There are six cases:

1. An empty block returns an emptyBlockResponse type. This is assumed to be the list of no interfaces as an anonymous termination.
2. A singleton returns the type of the single expression.
3. [; ;] The type of a sequential, last-value block has named terminations which are the meet of all the named terminations of all the expressions to the left of and including the first expression with no anonymous termination. It has an anonymous termination only if all the expressions have anonymous terminations, in which case it takes the anonymous termination of the last. e.g. given a block `[ a; b; c]`: if `a` and `b` both yield anonymous terminations, then the type is the meet of all terminations of `c`, and the named terminations of `a` and `b`. If `b` does not yield an anonymous termination, then the type is the meet of all terminations of `b`, and the named terminations of `a`. However it would be appropriate to generate a warning here since `c` can never be executed.
4. [| |] A last-value block with anything other than sequential execution has named terminations which are the meet of all the named terminations, and an anonymous termination only if all the expressions have anonymous terminations (in which case it is the last).
5. (; ;) A sequential all-value block has named terminations which are the meet of all the named terminations of all the expressions to the left of and including the first expression with no anonymous termination. It has an anonymous termination only if all expressions have anonymous terminations, in which case it is the concatenation of these. If no anonymous termination is found at any point, then the result is an early completion with the meet of the named terminations found so far.
6. (| |) An all-value block with anything other than sequential execution has named terminations which are the meet of all the named terminations. It has an anonymous termination only if all expressions have anonymous terminations, in which case it is the concatenation of these.

4.4 Body

operationBody → *block* | **code** [*language*] *embedded*

The type of a body will be the type of the block, or “no type” if the body holds code.

4.5 Declaration

declaration → *name* { *name* } **:** *typeExpression*

The type of a declaration is that of its type expression

Various functions may be required in an implementation of type inferencing. These include facilities to set a name into the symbol table along with its type, and to look up a name and retrieve its type.

The type of a declarations list is an interface list consisting of the types of each declaration in the declaration list.

4.6 HandledBlocks and Handlers

handledBlock → **after** *block* **handle (** { *namedHandler* } [*defaultHandler*] **)**

A handled block consists of a block and a list of handlers (for terminations), one of which may be the default handler (termination).

The block may return an anonymous termination and a list of named terminations, some of which may be dealt with by the named handlers. If there is a default handler, it will deal with any remaining named terminations.

Take first the case where there is no default handler. Named terminations generated by the block will be dealt with by handlers where these are provided, and any other named terminations will form part of the type of the block. In addition the handlers themselves may generate anonymous and named terminations. The anonymous termination of the handled block therefore comprises the anonymous termination of the block and the meets of the anonymous terminations from the handlers. The named termination will consist of those named terminations from the block which are not handled, and any additional named terminations generated by the handlers.

If there is a default handler, then this will handle all named terminations from the block. The anonymous termination is therefore as above, but the named termination consists of the meet of named terminations from all (including default) handlers.

Each handler consists of a termination name, a declarations list, and a block. These must be compared with the list of named terminations in the response type of the handled block, and checked for consistency. The termination names in the response type of the handled block may be in any order.

4.7 Identifier

Names have types associated with them. These are referenced from the symbol table.

4.8 Interface

interface → **interface** [*attributeList*] [**data** [*language*] *embedded*]
({ *signature operationBody* } **)**

The type of the interface is the “sum” of the types of its operations. The inferensor must also check the body types against signatures.

4.9 Invocation

invocation → *unit* **.** *operationName block*

The type of the unit must include an anonymous termination which is a single interface. If there is no anonymous termination, or if the anonymous termination consists of a list of more than one interface, then this is an error.

Having established that the type of the unit includes an anonymous termination with a single interface, then the operation name must be a valid operation on that interface. The signature of the operation provides a list of its formal arguments. The actual arguments given in the block must conform to this list of formal arguments, i.e. there must be the same number of arguments, where the actual types conform to those of the formal arguments.

The type of the invocation is the meet of the following:

- the operation was successful, giving the result type of the operation
- the evaluation of the unit produced a named termination, giving the type of the unit, less the anonymous termination
- the evaulation of the block produced a named termination, giving the type of the block, less the anonymous termination

4.10 Object

object → **object** [*attributeList*] [**data** [*language*] *embedded*] *block*

The type of an object is the type of its block.

4.11 Operation

signature operationBody

The signature must have at least one termination, although it need not necessarily have an anonymous termination. The type of an operation is its signature, but the evaluation of the body in environment of its arguments must yield a response type which conforms to the signature response type.

4.12 Signature

signature → *operationName* [*attributeList*] *argumentList*
-> [*terminationName*] *resultList*
{ **->** *terminationName* *resultList* }

The signature must have at least one termination, although it need not necessarily have an anonymous termination.

The type of the operation has two parts:

- the names and types of the formal parameters of the operation
- the type of the anonymous and named terminations

4.13 Terminate

terminate → **->** *terminationName block* | **-> reterminate**

The terminate expression generates a termination type whose name is the termination name. If the reterminate clause is used, then this generates a termination which is the same as that which is being handled.

4.14 TypeConstructor

typeConstructor → **type** [*attributeList*] **(** { *signature* } **)**

The type of a type constructor is the type of the interface resulting from the type of the signatures. This interface type is the sum of the operation types.

4.15 TypeDefinition

This is similar to a definition, but the name is that of a type, rather than an interface. Care must be taken to take any circularities into account.

4.16 TypeExpr

typeExpression → *typeName* | *typeConstructor* | *typeBlock*

This will take the type of either the name, constructor or block.

4.17 Unit

unit → *name* | *invocation* | *block* | *object*

The type of a unit is the type of the name, invocation, block or object which comprises the unit.

References

[ABAD89]

Dynamic Typing in a statically typed language Abadi, Cardelli, Pierce and Plotkin ACM symposium on Principles of Programming Languages, 1989

[ABAD91]

Dynamic Typing in a statically typed language Abadi, Cardelli, Pierce and Plotkin ACM TOPLAS 13 (2), 1991

[ANSA 93]

DPL Programmers' Manual, **TR.031.00**, APM Ltd., Cambridge U.K., April 1993.

[APM89]

The ANSA Reference Manual, Volume C, Release 01.01; , APM Ltd., Cambridge U.K., July 1989.

[APM91a]

The ANSA computational model; **AR.001**, APM Ltd., Cambridge U.K., August 1991.

[CANN89]

F-bounded Quantification for Object-Oriented programming Canning, Cook, Hill, Mitchell & Olthoff ACM conf. on Functional Programming and Computer Architecture, 1989

[Cardelli85]

Cardelli, L. and Wegner, P, *On Understanding Types, Data Abstraction and Polymorphism*; ACM computing surveys 17(4), December 1985, pp 471-522.

[Howarth91]

Howarth, N. J., Rees, R. T. O., Watson, A., *The ANSA Programmers' Manual*; **APM/RC.105**, APM Ltd., Cambridge U.K., September 1991.

[Howarth94]

Howarth, N. J., *TINA-DPE AST Design*; **APM.1180**, APM Ltd., Cambridge U.K., June 1994.

[Hutchinson87]

Hutchinson, N., *Emerald: An object-based language for distributed programming*; University of Washington Dept. of Computer Science Technical Report 87-01-01, 1987.

[KATI92]

Subtyping F-bounded types Katiyar Position paper for the ANSA workshop on F-bounded quantification, 1992

[ODP95]

Reference Model for ODP-Architecture, ISO, ITU Geneva, 1995.

[PIER92]

Bounded quantification is undecidable Pierce ACM symposium on Principles of Programming Languages, 1992

[Watson92]

Watson, A. J., *Revising the DPL type system*; **APM/RC.339.02**, APM Ltd., Cambridge U.K., June 1992.