Security Through Type Analysis

C O'Halloran and C T Sennett

Systems Engineering and High Integrity Systems Division
DRA Malvern
Worcs WR14 3PS
UK
email: colin@green.dra.hmg.gb, C.T.Sennett@green.dra.hmg.gb

Abstract. The objective of the work reported in this paper is to develop very low cost techniques for demonstrating that the trusted software for a secure system has the security properties claimed for it. The approach also supports integrity properties. The approach is based on type checking, which ensures that operations cannot be called with arguments they should not handle. This paper presents an informal technical description of the work with respect to a particular case study. An outline of the type checking algorithm is given in an appendix.

Keywords: Types, formal techniques, secure computer systems, security evaluation.

1 Introduction

This paper reports a new approach to establishing the security properties of a system, based on software analysis and type checking. The approach is motivated by the perception that current practice is leading to systems which are unusable, slow and costly to develop and maintain. At the heart of these troubles is the reference monitor concept: security is seen as being concerned with controlling access to objects according to clearances and labels, to be enforced by a reference monitor, which is software executing within an isolated hardware protection regime. This very simple view leads to performance problems, as all accesses to objects must be mediated by the reference monitor, and to usability problems as a result of the very simple controls which can be implemented in a centralised piece of software.

In addition to these usability problems, the reference monitor concept leads to a large cost of ownership because of the difficulty of disentangling the reference monitor from the operating system which it uses. The basic operating system primitives, above which the reference monitor runs, tend to execute with more privilege and could corrupt the reference monitor. To gain assurance that the security mechanisms work correctly it becomes necessary to evaluate all the security relevant code, now including the basic operating system primitives. This leads to the need to evaluate megabytes of code which is costly, and the inability to change the system while maintaining the evaluation status.

The desire for increased flexibility leads almost inexorably to the need for protection mechanisms based on software rather than hardware. Clearly there are many mechanisms such as data hiding, type checking and the use of modules which are highly relevant to security and potentially offer much more flexible protection than can be provided in hardware. In addition, being implemented at compile time, they have little or no run time overhead. However, just because these mechanisms are implemented by compilers and, in the case of languages such as C, can be subverted by the programmers, there is an integrity issue. How trustworthy can software be if the protection is derived from compilers, which are large and complicated programs? The approach we are developing uses an analysis technique, based on type checking, and applied to compiler output, to give an independent check of the integrity mechanisms in the compiler and it is hoped that this would lead to a technique for guaranteeing security without the need to centralise all the security checking code.

The need to maintain and update software components fits naturally within this type of software checking regime. What is required is that security properties should be broken down into the conditions which each individual software component must satisfy, in order to provide security for the system as a whole. These conditions must be expressed in terms of the software interfaces in order to give a test which can be applied during maintenance and it is hoped that this information, namely the software interface specification, can be used to drive the analysis process.

It should be emphasised that the work described here is very much ongoing. The analysis method is being specified formally and is at a fairly detailed level of development. The means of actually specifying interfaces to capture security properties has not been completed and to some extent is dependent on the analysis method. Nevertheless, the paper gives an overall view of the whole process and includes a case study to illustrate how it is expected to work.

2 Security Properties

Security in a system is achieved by a mixture of functional and non-functional properties. The functional properties are concerned with the correct operation of the security checking mechanisms while the non-functional ones are concerned with the absence of by-pass and side effects.

Functional properties are demonstrated by formal verification, but this will rely at some point on the integrity of the implementation language and the compiler which implements it. The desire to have a reliable basis for compilation has led to previous work with Ada [1, 2], but the desire to use commercial components leads to the necessity to establish integrity in software written using C. In either case one would like a check that what had been verified in the source language could be relied on in the machine code, but with C there is an additional problem in that the programmer is not forbidden in the language from misusing pointers in a way which would render the verification invalid and which could

corrupt other software. The analysis method supports this integrity check, not the verification which precedes it.

Non-functional security properties are often simply a question of capability: for example, the users should not be able to use the system management functions or it should not be possible to log in from a remote computer. These are naturally provided in a system which constrains the code only to use the interfaces explicitly provided. This works for static properties where it can be checked that one module simply cannot call another module providing the capability which it is required to control. However, the absence of by-pass is usually specified in behavioural terms as constraints on the sequence of operations allowed. One of the problems of specifying security is that the condition for an operation to be legitimate or not is dependent on what has gone before. The desire for flexible security controls means that these constraints can be quite complicated and so the interface descriptions must be able to reflect this.

2.1 Rely and guarantee conditions

If a system is composed of software components, they can be evaluated separately if the properties they rely upon can be determined. These rely conditions on the environment are "preconditions" under which the component will guarantee to exhibit some property. The guarantee condition is analogous to a post condition.

Rely and guarantee conditions were originally formulated to reason about safety properties of shared memory. They have been proposed as a means of proving confidentiality, but because of their origin are probably more suited to demonstrating absence of bypass of security mechanisms.

Absence of bypass of security mechanisms is the non-functional property required to demonstrate security. In behavioural terms, this is a safety property because absence of bypass can be shown when no illegal operations are used. The legality of an operation will depend upon what operations have occurred and thus absence of bypass is about demonstrating certain traces (that is, sequences of operations) are not present, which is a safety property.

A rely condition will be an assumption about traces which the environment is allowed to perform. A guarantee condition will be a promise to the environment to perform only certain traces. Put another way a rely condition states that the environment will not do anything to bypass the security mechanisms. A guarantee condition makes a promise to the environment not to bypass the security mechanisms.

A software component can be slotted into a system if the other components guarantee the behaviours the component is relying upon and it guarantees the behaviours the rest of the system is relying on. In this way components can be maintained and upgraded while still maintaining the property of absence of bypass.

2.2 Information flow

One of the problems with controlling the information held in a computer system is that information is itself a nebulous entity. Although it is useful to think of files and records being held by a computer system, they are actually just bits in a machine which can be duplicated and moved with great speed and ease by a program. The exploitation of indirect means of communication within a system is called using covert channels.

The problem of covert channels arises because the very mechanisms used to protect data can be used to communicate that data to unauthorized individuals. The covert channel problem, although a real threat, has been given undue prominence in recent years leading to over restrictive security properties which impair the functionality of the system.

The absence of covert channels looks at first sight like absence of bypass which it has been shown is an absence of illegal operations, assuming the integrity of the legal operations. Unfortunately covert channels arise from the legitimate use of legal operations. This means that the elimination of covert channels would eliminate useful and sometimes vital functions of the system: for an example, see [3].

To address confidentiality, it first has to be defined. In their seminal paper [4], Goguen and Messeguer gave a formalization of a confidentiality property in terms of observations of a system. The intuitive idea behind the formalism was that if a "low" observer cannot detect any change in the behaviour of a system when a "high" user is removed from the system, then that system protects the confidentiality of the "high" user. The "high" user is said to *non-interfere* with the "low" user; that is there is nothing the high user can do to communicate, covertly or otherwise, with the low user.

The idea of using observations to define confidentiality has been developed by Jacob [5] to give a formal meaning to the amount of confidentiality required. This is formalised over the trace model of CSP [6] and defines the degree of confidentiality to be the set of traces which can be inferred given full knowledge of the CSP process and a local observation. The smaller the inference set, the more certain a local observer is that a particular trace of the process has occurred. The bigger the inference set, the more secure a process is for that local observation.

The concept of using inference from a local observation of a system can be used to define a specification language for confidentiality requirements, [7, 9]. To show that a system satisfies such a specification it is necessary to show that it is meaningful to view the system as a process which engages in events. This means that the integrity of the events, or operations, which make up the traces of the system have to be ensured.

2.3 Type checking

The flow specifications give constraints on the sequence of events a given process can engage in. In a process description simple events have no specific meaning, they are just markers for the start, or end, of an operation or even the whole

operation. Events which have values associated with them carry more information about how they can be interpreted but they are still fairly abstract. In both cases there is an obligation to demonstrate a correspondence between software operations and the events they implement.

Type checking is one way of demonstrating this correspondence. Types give a meaning to the events of a process and allows a consistency check for a software interface against a process description. The types of values input from the environment can be used in type checking the software hidden by the interface in order to establish its integrity. Values which are output to the environment have a type which is determined by type checking the software hidden by the interface. This link between process events and operations on the interfaces of a software component are as yet unformalised.

Type checking in itself may provide one way of separating events and therefore could demonstrate the satisfaction of a flow specification. This would be the case for example if objects were given types according to their classification and separately typed procedures for handling them were provided. This possibility is unlikely to be generally useful as the flow properties are dynamic and specify constraints on sequences of operations. It is necessary for example, to make the results of a file opening operation depend upon the results of previous operations.

Such dynamic behaviour requires the storage of additional information to record the history and this information needs to be bound into the objects being controlled. Typically a file needs its classification, a process its clearance and user identity. Correct manipulation of this historical data is a functional property necessary for security. The functionality is usually trivial, but it could be established by verification if necessary. The non-functional property of lack of by-pass is that data can only be altered by the security checking code and it is this property which can be achieved by type checking.

Our approach is to treat such encapsulated objects as the implementation of an abstract data type. If the using code has correctly treated such objects as abstract, it should always have dealt with them as a whole. An abstract object can be used, assigned, provided as a parameter to a procedure and delivered as a result, but the internals of the object cannot be inspected or altered.

3 Software analysis and type checking

To summarise, our approach is to use flow specifications which are sufficiently flexible to capture the complexities of real requirements; to break these down into specifications for individual software components; to relate these specifications to the software interfaces using functional control of abstract objects; to provide lack of bypass at the source code level using the structuring and type checking properties of languages (and any other means relevant at that level); and to replace the hardware protection by an integrity check on the compiler. This is clearly a large scale programme, and one would expect to use many different techniques, particularly at the language level. We have concentrated on the last

problem, the demonstration of the integrity of the compilation process, as the one on which the others are built.

3.1 The integrity problem

A compiler transforms a source language into machine code: the integrity requirement is that the semantics of the source language are respected by the executing machine code. Basically, there are three issues:

- The mapping of operations into corresponding machine code operations (such as arithmetic and logic).
- The mapping of identifiers and objects in the source code to machine memory.
- The mapping of program structure into corresponding machine code structures (conditional statements and loops into corresponding machine code statements and jump instructions).

The third is clearly the most difficult as the mapping is not by any means one-to-one, nor is the same mapping used in the same syntactic situation. However, from the integrity point of view, one is concerned about the usage of objects defined in an interface. What is required is that the code which uses objects in the interface should not, by virtue of operations on its own identifiers, corrupt the interface. For this aspect, the first two elements are important.

The complication in these elements arises from the following sources:

- The machine operations are not defined on the natural numbers, but on several different formats (single length, double length, characters, floating point) which not only have different sizes, but also must be aligned at particular positions within memory.
- Machine manipulation of objects is usually in terms of pointer arithmetic. Pointers are not constrained to point to one particular type of data structure.
- A given memory location may hold, at one location, a value of one size at one time and a value of another size at different time, depending on the dynamic conditions of execution.

These last two points are particularly important for the compilation of languages such as C where pointer manipulation, and the use of unconstrained unions, are features of the language, rather than being introduced by the compilation process.

Rather than analyse machine code, we have chosen to use the TenDRA portability technology [8] and analyse its distribution format (TDF). TDF is a compiler output language (that is, the output from a compiler front end) which is capable of representing all the commonly used languages and can be translated into all the commonly used machines. The production of TDF corresponds to the code generation phase of compilation. Checking at this particular stage of the process has a number of advantages:

- Because it can be installed on a number of different machines and architectures, the TDF must make explicit all the interfaces. A correctly installed unit of TDF code cannot use any other unit, or make use of the machines operating system, except through the interface mechanism provided. This allows both the checking of the interfaces actually used and their replacement, at installation time, with equivalent interface components of equivalent functionality but with extra dynamic checking.
- The architecture neutrality ensures that one checking tool will suffice for several different machines.
- The checking process can add more TDF code to implement dynamic checks.

Type checking at the TDF interfaces allows the software object to be checked against typed events in a process description of the software object. This means that the alphabet of a process can be compared with the typed operations of a software object, this constitutes a static compliance check. Comparing the dynamic behaviour of a software object against a process is a more difficult task which is not addressed by this work.

In designing a checking process for TDF, the first obvious candidate is to use simple static type checking. This can establish things like the fact that an integer in an interface is only operated on by integer operations and a procedure is only called, not operated on by any other operation.

The integrity of the memory is rather more difficult to establish. Memory is accessed by taking the contents of a pointer, or assigning to a pointer. Pointer values are either given by an original allocation or derived from original pointers by pointer arithmetic. This involves the symbolic addition of offsets given in terms of alignments and the sizes of primitive objects. In checking the integrity of this process, the typing is being applied not so much to the pointers themselves as to the memory being pointed at. Pointer arithmetic constrains memory to have objects of certain sizes at certain points while assignment achieves the typing of memory at the position currently being pointed at. Type checking within this context is naturally done by inference.

3.2 Type checking

The actual layout of memory is going to be a compiler choice, rather than being specified by the user. What is required is that when an object is accessed in an interface, sufficient space is allowed to store it: this space must not be infringed by other objects internal to the using code. The type checking therefore has to deduce the layout being adopted by the compiler and make sure that that is being used consistently and that it respects the integrity of the objects imported from or exported to the interface. The method chosen for this is a system of type inference which is rather analogous to polymorphic type checking in the language ML. Here the memory is treated as polymorphic and the inferences establish the particular instantiation chosen by the compiler. As a by product of this approach, there is no problem in dealing with polymorphic procedures.

The actual type checking process for memory is almost statically determined. Apart from pointers to arrays, which correspond to original allocations, pointers will be used for accessing structures or equivalent operations. Pointer arithmetic within the structure will be determined by the layout of the structure itself, which must be a static quantity. Pointer arithmetic can therefore usually be evaluated statically and used to check the memory layout without the need for a dynamic check. The situation is rather analogous to a type system containing integer ranges rather than integers (for example, those for Ada or Pascal). In these languages it is often possible to check the types statically because the construction (a loop for example) will ensure the controlled variable will not go out of bounds.

The existence of unconstrained unions does unfortunately almost inevitably lead to dynamic checks. Within C, it is possible to construct a pointer and access from it at one time an integer, at another a floating point number. Provided the program previously stored an integer in the one case and a floating point number in the other there is no problem. Clearly, two consecutive accesses like this cannot be correct; assignment statements may satisfy them, but where the two cases are combined in a conditional, there is a dynamic condition to be propagated back to where the assignments were originally done. This part of the checking process is therefore rather like a weakest precondition calculation. In favourable situations it might be possible to simplify the well-typing condition to true. This might be the case for checking output from an Algol 68 compiler, which is strongly typed, achieved by the compiler outputting dynamic code to keep track of the current types holding for a union. Compiler generated code for these situations ought to be small, regular, and easily verifiable. Output from a compiler for a statically typed language, such as Algol 60, should not generate any well typing conditions at all. Where the user is in control of the dynamic conditions for the types, as in C, the conditions might be more complicated.

Thus the process actually chosen to demonstrate the integrity of the compilation is a combination of static type checking, type inference and a weakest precondition calculation for the dynamic checks. It should be emphasised that this checks the integrity of the compilation process. In particular, if aliasing is allowed in the language (as in Ada or C) this will not be detected. But if aliasing is forbidden in the language (as in the Ada subset SPARK) the checking process will guarantee that it does not occur through a fault in the compiler.

4 The File Transfer Case Study

Two case studies are being undertaken within the programme: one concerned with file exchange, which demonstrates a very simple application of the approach; and one concerned with the use of X-Windows which is very much more complicated. The way in which the approach might be applied will be described for the file transfer case. We have not yet completed either the decomposition of the security properties or the analysis of the software.

4.1 Background

Interoperability is an increasingly important concern with secure systems. A straightforward application of standard security practice would treat the two interoperable systems as a whole and attempt to work out an appropriate security policy and assurance level for the combined system.

This approach goes against the philosophy of building a total system from smaller simpler parts and is rarely a practical proposition. The individual systems will have different security requirements, management, and capabilities for evolution. The combined system will have a large population of users with conflicting functional as well as security requirements which will be difficult, if not impossible, to satisfy.

In practice the degree of interoperability required is usually quite small, and considerably less than the full functionality of a system. By restricting the interoperability to the minimum necessary to meet the requirements, it is possible to design a system which presents little possibility for misuse and it becomes tractable to demonstrate this.

The transfer of files is often all that is required. We shall assume a very simple situation in which a high security multi-level system needs to send files to a low security dedicated system. The direction of transfer (from high to low) is not just a question of choosing a perverse example. In real life information (commands, for example) often have to be exported to less classified environments. One of the problems with current systems is that these real life requirements are being disallowed by inappropriate security policies. With this sort of requirement, a one way filter is inappropriate and the solution often proposed is to isolate the high computer by means of a guard computer, which provides a facility by means of which a watch officer can inspect all information flowing through the guard. Unfortunately, no amount of verification of the software in the guard is going to reduce the vulnerability of the watch officer blinking or not realising the significance of what is being displayed on his screen. A guard computer is nearly always applying an inappropriate check at an inappropriate place. Instead the flexibility allowed by the checking approach allows an appropriate check to be applied, without the need for a guard or a watch officer.

4.2 The file transfer security requirement

There are many possible scenarios for file transfer and it is possible to design appropriate security policies for most of them. We shall take the simplest possible in which files are sent from the high security computer to the low security computer (called A and B respectively) as a result of actions on A. That is files are pushed from A rather than pulled from B. In this case, an adequate security policy, that is, one which countered the potential threats to the system, might consist of the following four requirements:

1. No file whose security is greater than the level of B may be sent to B.
2. Files only pass to B as a result of the action of an identifiable user on A.

3. Only B receives files from A.
4. Only file transfer from A to B happens on A.

The first of these is a functional requirement, while the other three, being essentially concerned with bypass, are non-functional.

These requirements statements are easily linked to the threat. The first ensures that the handling requirements are respected, while the second allow the action to be audited and counter Trojan horse attack in A's untrusted software. The third and fourth counter attacks via the network. Clearly a selection of security requirements can be made according to the threat situation of the system. Note however that the second requirement can only be implemented on A.

4.3 The implementation

The implementation is shown in figure 1:

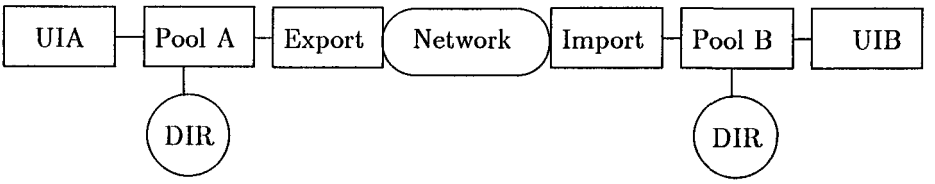

Fig. 1. File transfer diagram

The various components of this implementation are as follows:

UIA	The user interface on machine A. This allows the user to access files and give them to PoolA for sending to B.
PoolA	This checks the classification of files presented by UIA and if allowed invokes FileExport to send to B.
FileExport	This obeys the file transfer protocol at the instigation of PoolA.
Network	Carries out the data transfers between A and B.
FileImport	Carries out the receiving end of file transfer.
PoolB	Temporary store for files received.
UIB	User interface for storing the files received from A.

To implement the security requirements given above, only the network and the components on A need to be trusted. Their rely and guarantee conditions are as follows:

Network	**Guarantees:** A may only exchange data with B. **Relies on:** Encryption
FileExport	**Guarantees:** Provides file transfer facility only to PoolA. **Relies on:** Unique binding of network interface.
PoolA	**Guarantees:** Provides checked file transfer facility only to UIA. **Relies on:** Unique binding of FileExport interface.
UIA	**Guarantees:** File transfer only takes place as a result of user interaction. Whatever auditing and inspection functions needed. **Relies on:** The trusted path mechanism (probably another unique binding).

These have been chosen so that these trusted modules may be constructed from untrusted software together with a small amount of encapsulating code. The details clearly depend upon the features of the component, but one would expect to use the encapsulation of files with labels in PoolA.

Note that the rely conditions tend to invoke lower level mechanisms, such as encryption in the case of the network and configuration management in the case of those components relying on unique bindings of modules. It is not yet clear how to manage these conditions (for example, whether to formalise them, whether to store in a rule based fashion or simply in a database) in order to make the argument for security clear.

5 Conclusions

Making secure systems which actually work and which have defences appropriate to the attacks which might be made on them needs a significantly different approach from the access control and reference monitor approach which is commonly adopted. First of all, security policies need to take into account operational needs to transfer information from high security environments to low security environments. There is no operational point in having a system which only accepts information and gives nothing in return. This means policies must contain dynamic conditional elements.

These policies inevitably lead to security functionality being present in and enforced by software. This is true even of current reference monitor based systems. It is simply not possible to build a system in which all the security is concentrated in one place. This makes it highly desirable to be able to decompose security requirements so that the particular security requirements of a software component can be identified and checked during maintenance of the component.

In this decomposition process it is important to recognise that security has both functional and non-functional elements. The non-functional elements are usually associated with lack of side effects and bypass of the security mechanisms. The functional elements are relatively easily traced whereas the non-functional elements are not. Another important factor is that the decomposition usually involves change of representation as the design is implemented in high level language and the language is compiled into machine code. At these changes of representation, the issue is not just one of tracing the old requirement to the

new representation, but is also one of identifying the new hazards posed by the particular implementation decisions chosen. Appropriate counters to these hazards (such as configuration management and encryption) must be chosen and used effectively.

The particular hazards at the compilation stage are associated with lack of integrity in the translation process and errors in system construction. At the moment, the only counter to these hazards is to use a reliable compiler. An additional check, associated with the interfaces to a software component, would be highly desirable. This paper has described what such a check might be and how it might fit into the whole process of showing how a system satisfies its security requirements. This check, based on the application of type checking to compiler output, would allow a much more extensive use of software mechanisms to be made and would also allow for the incorporation of commercial or otherwise adopted software.

The checking process unfortunately cannot be entirely carried out at compile time: the elements which cannot be resolved may either be incorporated as dynamic checks or used as a measure of the quality of the software. It is possible that the checking process may be too strict for typical applications: further work will be necessary to resolve this issue.

References

1. B. A. Wichmann, *Insecurities in the Ada programming language*, NPL report DITC 137/89, National Physical Laboratory, Teddington 1989.
2. K. A. Nyberg, *The annotated Ada reference manual (2nd edition)*, Grebyn corporation, 1992.
3. C.O'Halloran, *BOOTS, a secure CCIS.*, DRA, Malvern Report 92002 1992.
4. J.A. Goguen and J. Meseguer, *Security policies and security models*, Proceedings 1982 IEEE Symposium on Security and Privacy, Oakland.
5. J. Jacob, *Specifying Security Properties*, in *Developments in Concurrency and Communication* C. A. R. Hoare, editor. The Proceedings of the Year of Programming Institute in Concurrent Programming), Addison Wesley, 1990
6. C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall 1985.
7. C.O'Halloran, *A Calculus of Information Flow (specifying confidentiality requirements)*, DRA, Malvern Report 92001 1992.
8. *TDF Specification*, available from Dr. N. Peeling, N101, DRA Malvern, St. Andrews Rd., Malvern, Worcs. WR14 3PS, UK.
9. C.O'Halloran, *Category theory and information flow applied to computer security*, DPhil thesis, Oxford University 1993.

Appendix

A An outline of the checking algorithm

A.1 Static type checking

As discussed in the main body of the paper, the checking algorithm consists of three parts: simple static type checking, a type inference system and a well-typing

dynamic condition generator. These parts are interwoven in the algorithm, but are described here separately. For static type checking, three things are necessary: the representation of the types, the abstract syntax of the language and the set of well typing rules which say how to give a type to the constructions in the language. The representation of the types is given by a sequence of Σ where:

$[\Pi, \mathit{Identifier}]$

$$
\begin{aligned}
\Sigma ::= \ & \mathit{Integer}\langle\!\langle \mathit{Variety} \rangle\!\rangle \\
| \ & \mathit{Floating}\langle\!\langle \mathit{Variety} \rangle\!\rangle \\
| \ & \mathit{Bitfield}\langle\!\langle \mathit{Variety} \rangle\!\rangle \\
| \ & \mathit{Offset} \\
| \ & \mathit{Pointer}\langle\!\langle \Pi \times \mathit{offset} \rangle\!\rangle \\
| \ & \mathit{Union}\langle\!\langle \Pi \rangle\!\rangle \\
| \ & \mathit{Proc}\langle\!\langle \Pi \rangle\!\rangle \\
| \ & \mathit{Named}\langle\!\langle \mathit{Identifier} \times \mathrm{seq}\, \Sigma \rangle\!\rangle \\
| \ & \mathit{Top} \\
| \ & \mathit{Bottom}
\end{aligned}
$$

Note that this is a simplification of the datatype actually being used as it omits parts necessary to take into account the alignment of items. The *Integer*, *Floating* and *Bitfield* constructors represent machine integers, floating point numbers and bit strings in various lengths. An *Offset* is the type given to values which can be added to pointers while *Pointer*, *Union* and *Proc* are used to represent values which are pointers, which can have several types, or which can only be called respectively. The *Named* constructor is used to implement the abstract data type concept and the *Top* and *Bottom* constructors are used to give top and bottom elements for forming LUB and GLB of types.

Missing from this datatype is any constructor corresponding to structure or cartesian product. In its place is the fact that a sequence of Σ (or rather a sequence of aligned Σs) is used for the types of expressions and memory. This allows for the free manipulation of pointers and the use of part structures, apart from the specific structure indicated by the *Named* types.

For pointers, unions and procedures it is necessary to know the type of what is being pointed at, the potential types for the union, and the potential types for the procedures. These are given using Π values, which should be treated as instance variables. Two pointers are assumed to have different types unless the Π values are the same, in which case it will be known that they have been derived from the same original pointer. Associated with each pointer Π-value there will be a sequence of aligned Σs corresponding to what has been inferred about the memory as a result of the usage of the pointer. The *offset* value, used in the construction of the pointer type, gives where in memory the pointer is currently pointing.

In a similar way, the Π-values associated with the union and procedure constructors allow information to be obtained from an environment about the conditions under which a union value has one of the types available to it and the type of a procedure value.

The abstract syntax of the language is given by the abstract syntax of TDF insofar as it applies to expressions and statements: both aspects of the language are contained within one very extensive data type called *EXP* which has well over a hundred constructors, corresponding to the various operations (arithmetic, assignment, conditional etc) which are available.

The checking process constructs a schema value:

$$\begin{array}{|l} \textit{Exp} \\ \hline e : exp \\ \omega : \Omega \\ \hline \end{array}$$

where Ω is the Z type for the sequence of aligned Σ and *exp* is an abstraction of *EXP* needed for the calculation of well typing conditions. The signature of the checking function for TDF expressions is therefore given by

$$check_{EXP} : EXP \times Env \nrightarrow Exp \times Env$$

That is, the checking function evaluates an *EXP* to an *Exp* within an environment, and may change the environment.

This function is defined by cases over the constructors of *EXP*: a typical example is the integer addition constructor which has two EXP arguments (and an error treatment argument). The two arguments should evaluate to integers of the same variety and this is the type of the result delivered.

A.2 Type inference

Type inference is used to deduce the layout of memory. The environment keeps the relation between the Π values used in the pointer types and what is known about the memory in the form of a sequence of aligned Σs. The inferences about the memory are made as a result of checking sub-expressions and working out their types. The use of Π values and the environment ensures that when an inference is made about a sub-expression it is propagated to the other elements of the expression.

A typical operation which allows inferences to be made adds an offset to a pointer. The offset is represented by a sequence of aligned shapes where the *Shape* datatype represents the size of objects. With some simplifications, this is given by:

$$\begin{aligned} \textit{Shape} ::= & \ \textit{int\_s}\langle\!\langle \textit{Variety} \rangle\!\rangle \\ | & \ \textit{float\_s}\langle\!\langle \textit{Variety} \rangle\!\rangle \\ | & \ \textit{bit\_s}\langle\!\langle \textit{Variety} \rangle\!\rangle \\ | & \ \textit{point\_s} \\ | & \ \textit{offset\_s} \\ | & \ \textit{proc\_s} \\ | & \ \textit{union\_s}\langle\!\langle \mathbb{F}\ \textit{offset} \rangle\!\rangle \end{aligned}$$

The sequence of shapes which make up the offset, is clearly related to the sequence of Σs which make up memory. If the memory is longer than the offset

then each of the shapes must correspond with the Σs. If the offset is longer than the memory, then the operation allows us to infer what the memory should be. New Σs are created corresponding to each of the shapes: where pointers, unions or procedures are involved, new Π values must be created.

To capture the effect of assignment on pointers, it is necessary to work backwards (that is, work out the pre-condition for well-typing). Consider an assignment such as

$$x := E;$$

where `E` is some expression. The type of `E` must be compatible with that of `x`: it might actually depend on `x`. However, what `x` points to after the assignment might not be the same as what it is pointing to before. This will only be the case if the assignment is in the path of a loop, so it is at the loop heads that expressions and pointers are unified, which in effect constrains the type of pointers to be invariant if they are used for assignment within the scope of a loop.

A.3 Dynamic type checking

This is required whenever the programmer uses unions, that is, where the same area of memory is used for objects of differing types. In order to calculate the well-typing condition, it is necessary to associate a condition with each of the possible types of a union and for each programme element, propagate this weakest pre-condition backwards. Given variables `r`, `i` and `u` of type (pointer to) floating, integer and a union respectively, a code sequence like

$$r := u;\ i := u;$$

is clearly incorrect. The two arms of a conditional may, however, use a union in two different ways. In the case of

$$\text{if } G \text{ then } r := u \text{ else } i := u;$$

where `G` is some boolean expression, the weakest precondition for well-typing is `G` for `u` floating, not `G` for `u` integer. This is often written as a predicate:

$$u\%Floating \wedge G \vee u\%Integer \wedge \neg G$$

Conditions of this form can be simplified at assignment. For example, the weakest precondition of `u := 3;` with respect to this condition is $\neg G$.