The Role of Programming Languages in the Life-Cycle of Safe Systems

Clemens Szyperski and John Gough
{c.szyperski, j.gough}@qut.edu.au

Faculty of Information Technology
Queensland University of Technology
Brisbane, Australia

July 26, 1995

Abstract

Safety as understood in the conference theme can be separated into the more technical terms of safety, progress, and security: nothing bad happens, the right things do happen, and things happen under proper authorization. All three interact to make a system "safe" in the broader sense. This article introduces to the degree of safety in the technical sense that can be directly supported by programming languages and their type systems in particular. From a generalized definition of type a brief journey through contemporary type systems is taken and illustrated using examples from different programming languages. Finally, current trends and some novel approaches are presented.

1 Introduction

With the ever increasing presence of computers controlling critical systems – critical to missions, the environment, human lives, or the society – the safety of such systems is a prime concern. Safety, as understood in the conference theme, is a rather general notion of "the system does what it should, and does not what it should not". A careful separation of aims is required in order to analyze in more detail how safety can actually be achieved and how it can be supported by technologies and methodologies.

There are different possibilities to split up safety into detailed aims. For example, within an economical framework one might want to consider the fact that "nothing happens that would severely exceed the budget". In a societal framework one might require that "nothing happens without a firm basis on a sound decision and responsibility process".

This article is focussed on the impact that the choice of using a programming language and its implementation has on the safety of the built system. For this particular domain it is helpful to separate the general term "safety" into the more technical terms of *safety*, *progress*[1], and

[1]Note that *progress* cannot be covered by *safety* unless time is handled explicitly. For example, in a hard real-time system, the safety requirement that actions meet their deadlines does imply progress. However, in many abstract formalisms, including those based on the predicate calculus, there is no notion of time: it is considered sufficient to specify that the right things happen "eventually", a notion that needs to be covered by specialized specification tools – such as temporal or real-time logic – and therefore justifies the separation of safety and progress.

security.

The informal meaning of the three categories is:

- *Safety*: Nothing bad happens.
- *Progress*: The right things do happen.
- *Security*: Things happen under proper authorization.

All three interact to make a system "safe" in the broader sense.

In some cases it is possible to statically enforce safety (or progress or security for that matter), by using construction methods that simply exclude bad cases. Examples are formal proofs of correctness or automatic derivation/construction of components using a proved transformer. In other cases it may be necessary to enforce safety dynamically: if something bad is about to happen, this gets detected and proper steps are taken. Clearly, and whenever at all possible, static safety is to be preferred over dynamic safety which is to be preferred over "safety by assumption".

More and more system components are manifest in software and programming languages are the most concrete tools of the software engineer; it is only natural to expect programming languages to help improve system safety. In fact, it is well known that languages - and in particular proper language paradigms and type systems - can do a lot: functional languages, languages with strong static type checking, and languages with statically enforced encapsulation mechanisms go a long way in helping programmers to get it right. Also, the gamut of powerful language concepts in favor of safety is still getting richer. For example, linear types [1] and behavioral subtypes [14], both a matter of ongoing research, strongly support static safety in languages with side effects.

Strangely enough, despite the existence of viable alternatives, many software components – including safety-critical ones – get produced using languages that either provide dynamic safety only, only partial static/dynamic safety, or no safety at all.

Some examples of programming languages in each of the above categories are:

- *Static and Dynamic Safety.* Oberon [22], garbage-collected versions of Ada[2], Java [23], Sather [20].
- *Dynamic Safety Only.* Smalltalk [11], Lisp.
- *Partial Static/Dynamic Safety.* Pascal, Modula-2, and Ada (using explicit deallocation).
- *Unsafe* C, C++.

A whole industry thrives on selling tools to fix the situation, debugging tools that help to uncover errors that should not have been possible in the first place, such as memory leaks or dangling pointers. In some situations static safety cannot be established (the compiler would run into undecidability problems), but dynamic safety should still be supported. There is no excuse to let an array index out-of-bounds violation pass!

A common argument against using languages of ultimate static safety, such as statically typed purely functional ones, is the observed inefficiency of their implementations, or much rather their

[2] Ada leaves garbage collection as an option to the implementation.

unpredictable performance[3]. There is some point to that and current research is aiming at the merger of imperative/object-oriented and functional paradigms. The anchoring concept are powerful type systems: the idea being that a type-correct program is safe with respect to clearly defined requirements. Current research aims at moving type systems forward from constraints expressed over individual variables to constraints expressed over related variables. Linear types are again an example.

In the case of static type checking all restrictions imposed by the typing are fully verified by the compiler; there is no run-time overhead whatsoever[4]. Advanced type systems, such as those commonly found in typed object-oriented languages, can render fully static checking impossible (undecidable) or at least impractical. In such cases some residual run-time type checks are needed[5]. Again, such dynamic checking should be used: there is no excuse to let a type violation pass!

The article is organized as follows. An informal introduction to type systems based on their historical evolution is given in Section 2, combined with an account of the role of types in programming. Examples from contemporary languages are given. Section 3 broadens the approach and introduces current and some novel ways of enhancing the expressiveness of type systems. Section 4 sketches safety-relevant applications of such type systems. Finally, conclusions and a summary of contributions is given.

2 Type Systems – Review and Brief History

A type system allows a programmer to express *restrictions*: invariant properties of a typed program fragment. Type checking aims at the mechanical verification of these invariant; static type checking establishes the invariants at compile-time, while dynamic type checking introduces run-time checks to verify at critical points of change that all invariants still hold.

A type system can be seen as specialized sub-language that allows to *annotate a program with invariants*. Most type systems are restricted to certain categories of invariants. For example, it is usually not possible to require that a certain variable always holds a prime number, i.e. "prime" is not a type in most type systems. If a type system is too restrictive it may become useless, of course, but it is important to understand that restricted expressiveness is unavoidable in a practical type system. It is easy to generalize a type system beyond the point of static checkability (sometimes this is necessary), but it is also easy to further generalize a type system beyond dynamic checkability. Clearly, a type system that cannot evan be dynamically checked is no more than a mechanism for the introduction of "formal comments", and should be avoided. Figure 1 illustrates the evolution of data types in programming languages.

Before going into further details it is important to fully understand the notion of type annotation. Most statically typed imperative/procedural programming languages require the programmer to almost completely type-annotate a program. Among the few exceptions are constants; e.g. in Pascal the actual type of the constant 123 is derived by the compiler – there is no need to write `CONST foo: INTEGER = 123`. Typed functional languages take the idea

[3]The quite acceptable average performance of modern implementations of functional languages is based on aggressive optimizations. If a particular programming style is not well covered by the transformations used to optimize functional programs, performance can suffer a great deal. The complexity of the transformations used makes it difficult for the programmer to understand performance implications of using one or the other coding style.

[4]In fact, the mere fact that static typing restricts a program opens doors for many advanced optimization techniques.

[5]A careful language design is necessary to ensure that the introduced dynamic checks occur at as few points as possible and that the checking points are obvious to the programmer.

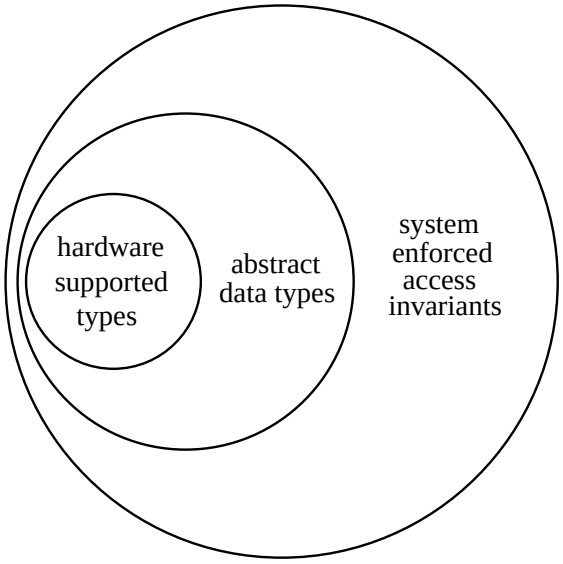

Figure 1: The evolution of type systems, outer rings come later

of compiler-inferred types further: inspired by principal type systems [12, 5], many functional languages allow for *partial type annotations*. The compiler computes the most general types that a given program fragment can deal with and raises a compile-time error if the result is inconsistent or ambiguous. Within the context of this article there is no need to further distinguish between fully and partially annotated programs, as long as it is guaranteed that the type inferencing has a sound basis, cannot lead to surprises, and does not try to use arbitrary heuristics for disambiguation.

A type system can in principle be used to annotate all *entities* and all *variables* of a program. Entities are constructed or primitive things of a language, e.g. constants, values, functions, or objects. Variables are named paths to entities, e.g. local or global variables, function parameters, or object fields. An entity can itself have variables, e.g. function parameters and object fields. Depending on the language at hand, variables may be constant or not and a single entity may be held by at most one, exactly one, or many variables at any one time.

A type is either identified primarily by its name or by its structure (its definition). Consequentially, there are two different approaches to telling whether two type expressions denote the same type: compatibility by name requires the same name, while compatibility by structure requires structural equivalence. Most imperative languages adopt compatibility by name, including Oberon, Pascal, Sather, Ada, and C++. In functional languages it is natural to think in terms of denoted values and therefore compatibility by structure is the common approach. It is also adopted by a few imperative languages such as Modula-3 [17]. Besides some advantages, compatibility by structure has a severe drawback: it can lead to *accidental compatibility*, a significant safety threat. Two user-defined types may be structurally identical, but have totally different meaning![6]

[6]To cope with such problems, Modula-3 introduces the compromise of *branded types*: such types are only compatible if they have the same structure *and* are labelled to belong to the same "brand".

2.1 Abstract Data Types and Modularity

Simple type systems go back to languages like FORTRAN: variables can be of one of the basic types of the language (e.g. integer or real) or of a constructed type (multidimensional arrays in the case of FORTRAN). ALGOL-like languages, and in particular Pascal introduced a rich set of type constructors to form records, sets, unions, and even files. These original type systems could be summarized to allow expression of invariants that restrict (specify) the domain of variables, i.e. a type is seen as *a set of possible values* that a variable of that type may be bound to.

The introduction of *abstract data types* (ADTs) led to a refined understanding of what a type is. Instead of viewing a type as a set of values, a type is viewed as an *algebraic sort*: a tuple defining a set of entities plus the operations thereon. The type definition lists the operations defined on the abstract data type. To form an algebraic structure, the type definition needs to be augmented by *axioms* that specify the semantics of the abstract operations.

value belongs to an abstract type
+ value belongs to a user defined abstract type
+ value belongs to a multi-sorted algebra (module)
+ objects belong to a behavioural (substitutable) type
...
= types as sets of entities defined by characteristic predicate

Figure 2: The evolution of abstract data types

Other than traditional constructed types, such as records, an ADT does not reveal its representation or implementation. Therefore, an ADT can be used to introduce a new type that behaves just like any built-in type. For example, neither INTEGER nor the ubiquitous ADT *Stack* reveal either their representation or the implementation of their operations. Furthermore, an ADT is free to impose restrictions such as limited *mutability*. The advanced type system of CLU [13] directly supports this by providing all type constructs in a mutable and an immutable variant.

To facilitate a clean separation of ADT interface and implementation, modular languages have been introduced. Modula-2 [29] and Ada [19] are the standard examples; Mesa/Cedar [16] and Modula [28] actually introduced the core concepts. It is possible to view a module defining and implementing a single ADT as being identical with the ADT. However, typical modular languages follow a different line: a module may well define (export) more than one ADT, and as well, it may partially or totally break with encapsulation by revealing part of the implementation details. Modula-3, for example, allows incremental revealing of implementation details: the idea being efficiency can be traded for encapsulation by picking an intermediate semi-abstract view.

The idea of using a module to present multiple interacting ADTs leads to the notion of *modules as multi-sorted algebras* or *modules as theories*. However, this again does not fully match the role of modules in modular languages. The prime role of modules is that of providing *confined contexts of analysis and system composition* [24]. For the purposes of this article this can be reduced to the requirement that just by inspecting the interface of all imported modules plus the implementation of the module under inspection, it should be possible to do static type checking. Finally, a module construct may provide for fine-grained access control of exported entities. For example, Oberon allows to export a variable *read-only*, while the module implementation itself remains free to modify the variables value. Restricted export interacts with the type system to enable programmers of modules to establish strong invariants without being concerned with the details of client modules.

The importance of the use of modules to decompose the task of establishing correctness cannot be overemphasised. The usual encasulation of ADTs ensures that only the encapsulated operations need to be shown to be correct in order to ensure that objects are always in a valid state. Similarly, the export modes of Oberon can ensure the validity of values even when the structure of the object is not opaque. Ada's limited private export mode goes even further, by supporting variables which outside their package are not only opaque, but cannot even be copied. In some contexts this is critical for establishing integrity properties. Notice that the separation of types and modules allows for the type invariant to depend on scope. Thus instead of a variable having the ANSI C attribute `const` everywhere, a variable may be constant except within a limited scope, or even stronger, is only subject to aliassing with a particular scope.

2.2 Genericity and Polymorphism

Concurrent with the development of modular languages attempts have been made to group and interrelate families of similar types. The two major threads are *generic typing* (parametric or bound polymorphism) and *subtyping* (inclusion polymorphism or object-oriented typing). Generic typing aims at factoring definitions and implementations by allowing type parameters. Subtyping aims at establishing compatibility relationships among types.

Generic types – generic packages in Ada or templates in C++ [9] – do allow for some degree of code re-use and therefore can increase safety[7]. However, the actual types resulting from different instantiations of generic types are usually incompatible.

When interpreting types as sets of acceptable entities it is easy to understand subtyping as forming defined subsets. In other word, a variable of a given type can be bound to entities of that type or of any subtype thereof. The key is to get a handle on the definition of useful subsets. For a subset to be useful it must fulfill a simple requirement: all entities in the subset must *behave* just as any member of the whole set, as far as this can be observed through the operations defined by the basetype [14]. In practice, no language exists that fully guarantees this requirement to hold. Most languages merely guarantee that a subtype has at least all of the operations of its basetype and that all these operations are defined using subtype-compatible signatures[8].

Subtypes have often been confused with sub-classes in early statically typed object-oriented languages[9], such as Simula [4], C++, or Eiffel [15]. Since subclassing (inheritance) leads to the inclusion of code, the requirement of behavioral substitutability for instances of subclasses is hard to establish. Clearly, an uncaught subtyping error can be a serious safety threat leading to arbitrary program misbehavior. Sather in turn fully separates classes and types [25]: types can only be derived (subtyped) from other types and classes implement types and inherit code from other classes[10].

Also, there have been proposals to integrate subtyping and genericity. Early attempts treated genericity as a special case of subtyping leading to languages like Eiffel. Later this has been refined by viewing genericity and subtyping as two orthogonal constructs, e.g. [21]. Consequently, Sather separately supports both, genericity and subtyping. In particular, generic parameters

[7] Relying on a single parametrized implementation for a series of related concrete implementations allows to amortize higher costs for specifying, verifying, and validating the shared code than would be possible for an unrelated set of repeated implementations.

[8] A compatible signature in a subtype must have covariant "out" and contravariant "in" parameters, e.g. [2].

[9] Surprisingly, Smalltalk does not suffer from this problem, as it simply does not have static type information at all: therefore Smalltalk classes are in a pure code inheritance relationship and it is only the programmer that might confuse subclassing with subtyping.

[10] Inheritance is viewed as mere textual inclusion, i.e. is a pure code factoring and re-use utility.

are bound by basetypes and therefore allow for static type checking of generic code. The latter is a known shortcoming of languages like Ada or C++: their generic constructs have unbounded parameters and cannot be type checked in general; each instance must be checked separately. Whether or not the type checking works out depends largely on the existence of the right operations in the parameter types. In such cases safety is reduced to the level of type compatibility by structure!

Finally, there have been attempts to unify classes and modules. Eiffel, once again, provides an example. Nevertheless, since the two concepts play quite different roles, it seems wise to have separate constructs instead [24]; Oberon or Ada-9X [26] provide examples of such separation. The final blending of module and type systems is still a topic of ongoing research.

2.3 Types and Exceptions

All properties of a system can break down if unexpected events – exceptions – cannot be dealt with properly. Exception handling is a long debated issue, but most discussion has centered on the semantics of the exception handling mechanism, on whether an interrupted computation is terminated or can be resumed, and on the granularity of exception handling[8]. Recent languages, including Modula-3 and C++, mostly converged on a termination model of exception handling with a termination granularity of a protected statement – following Modula-3 typically called a *try statement*. All of these language also provide for the automatic propagation of unhandled exceptions to the dynamically enclosing environment.

Languages in this category allow declaration of possible exceptions as part of procedure or function signatures. However, they typically allow a procedure that has been declared without any list of potential exceptions to raise any exception! The only language that got this point right, and also the first to introduce strongly typed exception handling, is CLU. In CLU a procedure simply cannot raise an exception that has not been declared in its signature. If code may raise an exception and it is not handled locally and it is not declared as a possible exception of the enclosing procedure, then a compile-time error is flagged. CLU implements a rather different model of propagation, which almost forces exceptions to be handled locally by the caller of the faulting routine.

It seems obvious that code that is not supposed to raise a particular exception should not be able to do so. Following this policy through adds some burden for the programmer and hence C++, Modula-3, and many other languages are less strict in their demands. As a result a programmer must either know the implementation of an interface (which breaks encapsulation, extensibility, and principles of modular development), or must be entirely defensive and expect and handle *all possible* exceptions whenever calling non-local code.

At a second glance it seems that there is a contradiction between strong typing of exceptions and extensibility: how can an extensible interface (say an abstract base class in C++) list all possible exceptions that may be raised by one of its methods? The problem can be solved by using inclusion polymorphism (subtyping) for exception types as well! For example, an abstract interface may declare that, say, operation *SendMessage* may raise exception *CommunicationException*. Then any concrete implementation of *SendMessage* is free to raise that exception or any subtype thereof, e.g. *UnknownReceiver*, if that is a subtype of *CommunicationException*. This facility is supported by all modern languages that cover exceptions, including C++, Modula-3, Java, and Sather. However, none of these languages (CLU again remains the exception) force a programmer to declare possible exceptions, leaving a safety gap open.

3 Going Beyond Sets of Entities

Although quite elaborate, all the type schemes introduced so far can be understood as defining potential *sets of entities*. (Recall that an entity is anything that a language offers and that a variable can be bound to.) The initial claim above was that type systems allow the expression of certain invariants over program fragments. By restricting a type system to look at a single variable–entity relation at a time, the possible invariants are limited to the same scope. However, consider the definition of a statically typed function: each of the parameters is associated with a type. Hence the function implementation can rely on the correct typing of all arguments[11] of the function. How does this fit into the model of individual and separate variable–entity relations? The answer is simple: the parameters are fully independent, just as a set of global variables would be, and the static matching of argument types against parameter types does not at all consider the combination of the parameter or argument types.

A more general view is that type systems can themselves be seen as little functional programming languages superimposed on their host languages [27]: In the case of static type checking the "type program" gets fully evaluated by the compiler. Advanced type systems, such as those supporting inclusion polymorphism as commonly found in typed object-oriented languages, can render fully static checking impossible (undecidable) and some residual run-time type checks are needed.

3.1 Multi-Methods

Inspired by the functional view of type systems one might have a look at dynamic function selection based on argument types. Object-oriented dispatch (late binding) usually uses the type of a single distinguished object, the *receiver*, to select the appropriate method to execute. In CLOS [6] this has been generalized to generic functions or multi-methods, but CLOS is untyped; Cecil is the first language to introduce fully typed multi-methods [3].

For example, consider the problem of defining addition for all numerical types supported by some language, say integer, real, and complex. All of these types support addition and since the mathematical fields they approximate are subfields of each other, mixed additions are defined as well and yield a result of the next enclosing field. In Cecil one might write —

[11] Arguments are sometimes called actual parameters; parameters are then called formal parameters.

```
add ( x @ complex, y @ complex )
-- x@T specifies dispatch on x
-- the specified method is selected if x is of type T
   ^ ... code to return the sum of two complex numbers

add ( x @ complex, y @ real )
   ^ add ( x, asComplex(y) )

add ( x @ complex, y @ integer )
   ^ add ( x, asComplex(y) )

asComplex ( x @ real )
   ^ ... convert real to complex and return

asComplex (x @ integer )
   ^ asComplex (asReal(x))
...
```

The main point is that code is written individually for each separate case. When writing add(x, y) somewhere in a program, the Cecil run-time inspects the dynamic types of x and y and uses this information to lookup the method that implements addition for these particular types[12].

Clearly, multi-methods eliminate many complicated case analysis constructs and thus seem to be a strong feature. However, reality is not that simple: multi-methods need an additional harness to become useful. The problem becomes visible when combining multi-methods with modules. Multi-methods effectively implement a table lookup in a fully populated Cartesian product: each parameter of a multi-method introduces a separate dimension for subtyping and for multi-methods to work it is necessary to fully inspect all possible combinations. It may not be necessary to implement separate methods for all combinations, since a single method can handle multiple combinations, but it needs to be checked that each combinations is indeed and without ambiguity handled by exactly one method.

For modules to be useful it must be possible to analyze each module separately. If definitions of subtypes can cross module boundaries, the presence of multi-methods makes separate checking impossible. There are two possible conclusions: restrict a multi-method to a single module or introduce a new construct to "seal off" a set of modules and prevent further addition of subtypes outside of the set that would affect the selection of multi-methods in the set. Cecil takes the latter approach and introduces *resolving modules* that effectively form the closures of the before mentioned sets of modules.

3.2 Linear Types

Instead of focussing on a set of entities when trying to form more powerful typing constructs, it is also possible to stay with a single entity but consider multiple or even all possible variables that might be bound to that entity. A type construct along this line that is attracting research attention are the *linear types*. Linear types are based on linear logic [10], for a introduction to linear types in the context of contemporary languages cf. [1].

A linear type restricts the number of variables that can be bound to a (linear) entity at any one time to exactly one. In other words, once created, a linear entity is held by exactly one

[12] In general, Cecil uses the most specific method closest to the actual types. Ambiguities are considered errors.

variable at a time and explicit destruction of the entity is the only way to get rid of it again. Linear entities have the interesting property of never being accessible via multiple paths, i.e. there is no aliasing and therefore no need for synchronization.

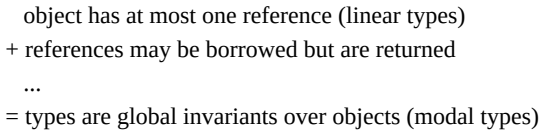

Figure 3: Types with access or sharability invariants

The use of such types is well established in functional programming, but is still the subject of research in the imperative setting. The attraction of such types include the possibility of safely using very aggressive compiler optimisations, the simplification of dynamic memory management issues, and the elimination of mutual exclusion concerns from distributed or multi-threaded programs. Implementation issues are not yet well understood however.

3.3 Modal Types – Generalizing Linear Types

As indicated above, linear types depart from traditional classes of predicates expressable by type systems in that they simultaneously cover all possible variables that potentially could be bound to a given entity. Linear types restrict the number of simultaneous access paths to an entity. This idea can be followed further to introduce other global restrictions on how entities can be accessed.

It has been suggested that linear types within an integrated language can allow for easy handling of mutable single-owner objects, before they get published as immutable shared ("functional") objects.[1]. This possibility hints in an important direction: the abolition of separate access control mechanisms in multi-threaded environments in favor of new type systems.

The *Gardens Project*[13] [7] is an example of a current research project developing new type concepts driven by the idea of generalizing linear types. These new types are called *modal types*, a term which subsumes the concept of linear types, but adds some novel type concepts as well, e.g. *borrow types*.

The key idea of a borrow type is to "lend out" a reference to an entity – thereby temporarily losing all access – to some other variable with the additional requirement that as soon as that other variable ceases to exist, the reference will be transferred back to the original owner. This concept requires the use of variables with a controlled lifetime, such as parameters of procedures or variables introduced by *let statements*.

At first glance it seems that the concepts of linear and borrow types overlap: with linear and with borrow types the original reference is lost when passing it on. However, with borrow types the reference is guaranteed to return[14] and the referenced entity is guaranteed to still exist. Also, a borrow type by itself says nothing about the number of references that might temporarily exist before the lent-out reference is returned.

As an example, consider a simple database serving clients that execute in parallel. In a traditional setting, items "checked out" from the database need to be locked to prevent *write-*

[13]Gardens aims at enabling parallel programming across networks of workstations. One of the key objectives is to develop a new programming language that is tightly integrated with its supporting system to minimize overheads. An important part of that development is the careful evaluation of type systems to lift the level of expressible global invariants.

[14]In principle, even under the presense of exceptions! This adds some burden to a possible exception handling scheme in a language supporting borrow types.

write conflicts with other clients. Using a modal type system, the data base would call a client closure and pass the requested item using a borrow type to serve clients asking for exclusive write access. The concept of locking the item is replaced by the concept of temporarily removing it from the database under control of the type system. Of course, this does not eliminate the possibility of *deadlock*, but it may open ways for analysis methods to check for guaranteed deadlock freeness of some given program.

It is interesting to compare the combination of borrow and immutable types with the concepts of *single assignment variables*, plain immutable types, or constant types as known from Ada, C++, or ANSI C. With all of the latter types a computation can be performed once resulting in an entity that thereafter can no longer change. This can be very useful, but quite often turns out to be too strong. For example, the "owner" of an entity may still want to be able to change it: something that is supported by Oberon's read-only or Ada's limited private types. However, it is usually only safe to change such a semi-constant entity when "no one is looking", i.e. when no other references to the entity are currently active. That is easy to establish by combining borrow and immutable types: whenever a client asks for the entity, it is handed out *borrow, immutable*, while inside the owner module the entity remains mutable whenever it is accessible, i.e. when it is not lent out.

Since a variable of borrow type has to fulfill a strong requirement, it can only be used to pass entities on to other borrow types variables – all other uses would require global analysis to establish that the borrowed entity is indeed returned. This requirement is similar to that for linear types, but even stronger in that a linear entity can be freely destroyed by its current owner, i.e. a linear entity is always fully owned by the variable currently holding the reference to it.

Borrow types are related to the concept of *observer types*[18], again from the domain of functional languages. However, just as for the linear types, there is little known about the integration into imperative settings[15]. A thorough investigation of these matters is underway as part of the ongoing Gardens Project.

The novel concept of modal types promises to enrich the potential of type systems and to further strengthen their role in programming for safe systems.

4 Powerful Type Systems and Their Effect on Safety

Type systems which enforce useful invariants on their variables have the possibility to enhance the safety of software systems, and to assist in the task of analysing correctness. As type systems have evolved, so the range of program errors which it is possible to detect as type violations has widened. The avoidance of type errors in programs is so fundamental to safety, that it should be a matter of some incredulity that such a large proportion of current software depends on languages with weak type systems, and primitive mechanisms of encapsulation.

In some domains, the soundness of the type system is critical to the security of applications, as well as to correctness. Such is the case with the Java language, intended as a language for constructing executable content in the World Wide Web. For such an application, it is important to ensure that object code does not (for example) breach the type system and invalidly access data. If this were allowed, it would allow virus-like vandalware to be inadvertently loaded by users. Indeed in this case, although a correct compiler can only produce typesafe code, the Java runtime system cannot rely on this. In Java, downloaded object code is subjected to one last level of typechecking in order to guard against a rogue compiler which might try to trick the

[15] Or object-oriented settings for that matter.

system into accessing data outside of declared objects. Java is a C++ based language, however many constructs had to be removed, and the type system needed to be significantly strengthened to provide safety and security. A comparison of the two languages is an enlightening exercise.

In general, type systems in which type checking is fully static do not require any runtime type checks. However, they may require other runtime tests in order to guarantee their declared invariants. A simple example is the use of subrange types in Pascal. The use of well chosen subrange types in Pascal can almost completely eliminate index bounds checks. However, in order to guarantee that the values of such variables are valid requires runtime *range* tests. In almost all programs, this leads to a smaller runtime overhead, which in any case is exceptionally small. In the case of the more radical uses of type systems which are currently the subject of research, there will often be runtime overheads in maintaining declared invariants. However, elimination of other runtime tests may more than pay for the cost, as well as providing additional safety.

5 Conclusions

Type systems may be viewed as a mechanism to introduce assertions about the entities in a program. In the short history of programming languages one evolutionary path has been to introduce progressively more powerful type systems. In the case that the type system is either partly or fully statically checkable, a large class of incorrect programs may be detected and eliminated during compilation. Even in the case where type correctness is not completely statically determinable, such type systems provide a framework for analysis which guides the introduction of economical runtime checks to complete the type safety guarantee.

Statically checkable type systems play an important role in the safe use of software components. Without a sufficiently powerful type system, the type correctness of an individual component is not preserved by the incorporation of the component into a compound framework. A particular issue is the so-called fragile base class problem, in which the extension of a particular class invalidates the correctness of an existing component.

Modules provide an encapsulation mechanism which provides a physical embodiment of the logical decomposition of a complex system. In conjunction with strong static typing, modules provide a framework in which arbitrary compositions of typesafe modules are also typesafe, thus helping decompose the task of analysing correctness. Modularity is thus a key attribute which provides the framework for the construction of trustworthy software systems.

Future languages may incorporate some of the more exotic types which are currently the subject of research. These languages offer not only the promise of a higher level of automatic checking, but may also provide performance advantages to their users.

References

[1] H. G. Baker. 'Use-Once' Variables and Linear Objects – Storage Management, Reflection, and Multi-Threading. *SIGPLAN Notices*, 30(1), January 1995.

[2] L. Cardelli. Typeful programming. Technical Report TR–45, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1989.

[3] C. Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92), Utrecht, The Netherlands*,

volume LNCS 615 of *Lecture Notes in Computing Science*, pages 33–56. Springer Verlag, June 1992.

[4] O.-J. Dahl and K. Nygaard. Simula - an ALGOL-based Simulation Language. *Communications of the ACM*, 9(9), September 1966.

[5] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, January 1982.

[6] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the First European Conference on Object-Oriented Programming (ECOOP'87)*, volume LNCS 276 of *Lecture Notes in Computing Science*. Springer Verlag, June 1987.

[7] J. Diederich, J. Gough, G. Mohay, and C. Szyperski. The Gardens Project – an introduction. In *Bound Collection of Submissions to the First Australasian Computer Architecture Workshop (Adelaide, South Australia)*, pages 17–24. Not available. A softcopy can be retrieved from URL `http://www.fit.qut.edu.au/~szypersk/Gardens`, January 1995.

[8] S. Drew and K. J. Gough. Exception handling: Expecting the unexpected. *Computer Languages*, 32(8):69–87, August 1994.

[9] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[10] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[12] R. Hindley. The principal type scheme of an object in combinatorial logic. *Trans. Am. Math. Soc.*, 146:29–60, December 1969.

[13] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[14] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), November 1994.

[15] B. Meyer. *Eiffel – The Language*. Prentice Hall, 2 edition, 1992.

[16] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, Palo Alto, CA, April 1979.

[17] G. Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice Hall, 1991.

[18] M. Odersky. Observers for linear types. In *Proceedings of the European Symposium on Programming*, February 1992.

[19] D. of Defence. *Reference Manual for the Ada Programming Language*. United States DOD, Washington D.C., November 1980.

[20] S. Omohundro. The Sather programming language. *Dr. Dobbs Journal*, 18(11):42–48, October 1993.

[21] J. Palsberg and M. I. Schwartzbach. Type substitution for object-oriented programming. In *Proceedings of the Joint Fifth Conference on Object-Oriented Programming Systems, Languages and Applications and the Fourth European Conference on Object-Oriented Programming (OOPSLA/ECOOP'90)*, pages 151–160, October 1990.

[22] M. Reiser and N. Wirth. *Programming in Oberon – Steps beyond Pascal and Modula.* Addison-Wesley, 1992.

[23] Sun Microsystems. The Java Language Specification. Technical report, Sun Microsystems, May 1995.

[24] C. A. Szyperski. Import is not Inheritance. why we need both: Modules and Classes. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92), Utrecht, The Netherlands*, volume LNCS 615 of *Lecture Notes in Computing Science*, pages 19–32. Springer Verlag, June 1992.

[25] C. A. Szyperski, S. Omohundro, and S. Murer. Engineering a Programming Language – the Type and Class System of Sather. In *Proceedings, First Intl Conf on Programming Languages and System Architectures*, number 782 in Springer LNCS, Zurich, Switzerland, March 1994.

[26] S. T. Taft. Ada 9x: From Abstraction-Oriented to Object-Oriented. In *Proceedings of the Eighth Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 127–136, October 1994.

[27] S. Thompson. *Type Theory and Functional Programming.* Intl. Comp. Science Series. Addison-Wesley, 1991.

[28] N. Wirth. Modula: A programming language for Modular Multiprogramming. *Software – Practice and Experience*, 7(1):37–52, January 1977.

[29] N. Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer Verlag, 4 edition, 1988.

6 About the Authors

Clemens Szyperski is an Associate Professor in the School of Computing Science and Director of the Research Concentration in Programming Languages and Systems, Queensland University of Technology, Brisbane, Australia. In 1992/93 he was a postdoctoral research scientist at the International Computer Science Institute affiliated with the University of California at Berkeley. He received the PhD in CS in 1992 from the Swiss Federal Institute of Technology (ETH Zurich) and the Dipl.-Ing. in EECE from the Aachen Institute of Technology (RTWH), Germany, in 1987. In 1994 he co-founded Oberon microsystems Inc, Basel, Switzerland and contributed to the design and implementation of the Oberon/F component framework. His research interests are extensible and distributed systems and the programming languages to support these. He has been participating in the Oberon and Ethos language and system projects at ETH, as well as in the Sather language and Tenet realtime communication projects at ICSI. At QUT he heads the Gardens Project, aiming at the definition and implementation of an integrated programming language and system to ease the creation of safe and efficient parallel programs executing on networks of workstations. A major focus in the Gardens project is to fully utilize static properties of the language and the (trusted) compiler to minimize local (non-maskable) overheads.

John Gough is Acting Dean of the Faculty of Information Technology and Professor in the School of Computing Science, Queensland University of Technology, Brisbane, Australia. A PhD graduate of the University of Wellington in his native New Zealand, his interests were originally in hardware, but switched to software in the late 1970s. Since that time, he has worked almost exclusively on programming language implementation, and is well known for the "gardens point" family of language compilers. His current research interests encompass code generation and optimisation theory, particularly for modular languages. He heads the language implementation team in the Gardens project.