# Functional programming languages for verification tools: a comparison of Standard ML and Haskell

**Martin Leucker**[1],[1], **Thomas Noll**[2], **Perdita Stevens**[3],[2], **Michael Weber**[2]

[1] Department of Computer Systems, Uppsala University, Box 337, 75105 Uppsala, Sweden
e-mail: leucker@docs.uu.se
[2] Lehrstuhl für Informatik II, RWTH Aachen, Ahornstr. 55, Aachen, Germany
e-mail: {noll,weber}@i2.informatik.rwth-aachen.de
[3] School of Informatics, University of Edinburgh, JCMB, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
e-mail: Perdita.Stevens@ed.ac.uk

**Keywords:**

**Abstract**

We compare HASKELL with Standard ML as programming languages for verification tools based on our experience developing the verification platform TRUTH in HASKELL and the Edinburgh Concurrency Workbench (CWB) in Standard ML. We review not only technical language features but also the "worlds" of the languages, for example, the availability of compilers, tools, and libraries. We also discuss the merits and difficulties of comparing programming languages in this wide sense and support our view that TRUTH and the CWB are similar enough to justify the conclusions drawn in this paper.

## 1 Introduction

Concurrent software and hardware systems play an increasing role in today's applications. Due to the large number of states and to the high degree of non-determinism arising from the dynamic behaviour of such systems, testing is generally not sufficient to ensure the correctness of their implementation. Formal specification and verification methods are therefore becoming more and more popular, aiming to give rigorous support for the system design and for establishing its correctness properties, respectively (cf. [1] for an overview).

In view of the inherent complexity of formal methods, it is desirable to provide the user with tool support. It is even indispensable for the design of safety-critical concurrent systems where an ad hoc or conventional software engineering approach is not justifiable. For example, *model checking* is a particularly successful automated approach to verification in which one employs decision procedures to prove that (a model of) a system has certain properties specified in a suitable logic.

One major concern in the development of model-checking tools and other verification tools is *correctness*.

[1] Supported by European Research Training Network "Games"
[2] Supported by EPSRC GR/A01756/01

Since a verification tool is used for verifying hardware, protocols, and software, it would be useless if it were not trustworthy. Thus, a programming language employed for developing a verification tool should support the task of verifying the code, at least in the informal sense of satisfying oneself that the code is correct. In general, *functional languages* are often considered to provide this feature, especially if the language disallows *side effects*.

Considering the broad range of functional languages that have been designed, it is surprising, and in our view unfortunate, that there is little literature comparing different functional languages. The Pseudoknot benchmark paper [3] studies several implementations of functional programming languages (HASKELL and Standard ML among them) with respect to their runtime and memory performance, and [7] compares the module systems of HASKELL and Standard ML. However, the developer trying to choose between the languages needs to be concerned about a much wider class of issues, including both technical language features and "environmental" aspects such as the availability of libraries, documentation, support, and multiplatform compilers. We did not find good sources of help for a developer trying to choose between the languages based on a larger collection of relevant aspects like this. By contrast, many comparisons of Java with C++ are readily available.

It is unsurprising, therefore, that developers (those who choose a functional language at all) often choose the language most used in their institution, without seriously considering alternatives. The difficulty of getting information to guide an informed choice may also contribute to developers whose workplaces do not have a history of functional programming language use deciding against experimenting with one.

Why is there so little material to help developers make an informed choice? Part of the reason must be that it is very hard to do convincing comparisons of languages without being vulnerable to the criticism that one is not comparing like with like. We think that a fair comparison needs to be based on real experience of people using the languages to build real systems in the same domain; otherwise it is almost impossible to be sure that differences are not due to differences in the domains of application. The systems themselves need to be broadly comparable in size and complexity, need to be more than toys, and should preferably have been developed and maintained over years (since a language that makes development easy might nevertheless encourage the development of code that is unmaintainable). Moreover, the domain should be one where either of the languages is a reasonable choice, and the comparison should be done by people with a reasonably typical level of experience in the languages. A comparison is probably most generally useful to developers when it is done neither by novices in the languages compared nor by people intimately familiar with the compiler internals.

This paper recounts our experiences in using SML and Haskell for two broadly comparable applications in the domain of verification tools on which the authors have worked for some years: the Edinburgh Concurrency Workbench (CWB), in SML, and the verification platform Truth, in Haskell. The domain of *verification tools* is eminently suited to the use of a statically typed functional language such as SML and Haskell, and both languages are popular choices in this domain. All of the authors have accumulated considerable experience with the languages we use, but we are not functional programming researchers.

Thus our primary motivation for writing this paper is that we believe we are in an unusually good position to produce a comparison of Haskell with SML that may be useful to developers choosing between the languages. A secondary motivation is to be helpful to language designers and developers who work to support languages by providing a record of our experiences, good and bad, with SML and Haskell.

The rest of this paper is structured as follows. Section 2 discusses the domain of verification tools and introduces the two systems. Section 3 discusses the class of languages from which SML and Haskell are drawn and briefly introduces the two languages for readers who may not be familiar with them. Sections 4 and 5 are the main body of the paper; Sect. 4 compares Haskell and SML on the basis of their technical language design features, whereas Sect. 5 considers the equally important "environment" aspects. Finally, Sect. 6 concludes.

## 2 Verification tools

The domain on which we compare SML and Haskell is that of verification tools. The term "verification tool" covers any tool whose task it is to assist in checking the correctness of some artefact. Usually the artefact concerned is (an abstraction of) something produced in the software or hardware development process.

We introduce Truth and the CWB and briefly summarise their histories before discussing the characteristic features of verification tools in general.

### 2.1 The Edinburgh Concurrency Workbench, in SML

Work on the CWB[3] began in 1986. The CWB's key strength is its breadth: a variety of different verification methods are supported for several different process algebras. In particular, it allows users to:

- Define behaviours given either in an extended version of Milner's CCS (Calculus of Communicating Systems [5]) or in its synchronous version, SCCS, and to perform various analyses on these behaviours,

[3] `http://www.dcs.ed.ac.uk/home/cwb/`

such as exploring the state space of a given process or checking various semantic equivalences and preorders;

- Define propositions in a powerful modal logic and check whether a given process satisfies a property formulated in this logic;
- Play Stirling-style model-checking games to understand why a process does or does not satisfy a formula;
- Derive automatically logical formulae which distinguish nonequivalent processes;
- Interactively simulate the behaviour of an agent, thus guiding it through its state space in a controlled fashion.

One major focus of the CWB was always research; it was a platform which researchers (especially those at Edinburgh) could use to experiment with new relations between processes and new algorithms. In the early years all of these changes were retained in the main tool, even those which had been added experimentally without much consideration for the integrity of the CWB overall. This contributed to the architectural degradation of the CWB and its increasing fragility: an important task faced by Stevens on taking over the maintenance of the CWB in 1994 was to reverse this process. The current version of the CWB consists of around 25 kloc in SML, plus several thousand in other languages for various supporting utilities.

The CWB was developed in Standard ML, but variations were long maintained for several major ML compilers because different compilers provided different extensions to the SML90 standard, and especially because they had different system build facilities. We settled on Standard ML of New Jersey (SML/NJ) because most users of the CWB used that compiler and the effort in maintaining build scripts (the major point of difference) for several compilers did not seem well spent. Perhaps we should once again target Poly/ML,[4] for example, in future. We will discuss the history of the family of ML-like languages in Sect. 3. For now it suffices to say that this paper inevitably considers SML/NJ more than any other SML compiler, and that since our experience is with SML, we do not consider in depth other languages in the ML family, specifically Caml and O'Caml. The contribution made by the SML language to both the architectural degradation problem and its solution are discussed later.

[4] `http://www.polyml.org/`

## 2.2 The verification platform Truth, in Haskell

In terms of features, TRUTH[5] is similar to the CWB. In its current version, it supports tableau-based model checking for the full $\mu$-calculus and game-based model checking for the alternation-free subcalculus. Both operate on finite transition systems, given in terms of CCS processes. The latter can be visualised and simulated in an interactive fashion, to help the user understand TRUTH's answers. Current development activities concentrate on the parallel implementation of model checking on a cluster of workstations and on a specification language compiler generator which, given the definition of a language, automatically generates a corresponding parser and a semantic evaluator.

TRUTH's initial version dates back to 1997, and its development could benefit a lot from the progress made in the design of verification tools over the years. As a consequence, its architecture is quite modular and easy to understand, and a deep change of the module structure has not been necessary so far. It now consists of approximately 18 kloc in HASKELL. Although there are several HASKELL compilers, TRUTH is written for the Glasgow HASKELL Compiler[6] (GHC), and since it uses some non-standard HASKELL extensions and libraries only present in the GHC, we have not tried to port it.

Moreover TRUTH employs the parser generator HAPPY[7] and many of the available HASKELL libraries, and it integrates several stand-alone systems such as the DAVINCI[8] graph visualisation tool and the GraphViz package.[9] Furthermore, it uses existing C and Java libraries to provide functionality such as textual and graphical user interfaces and network communication, comprising approximately 13 kloc. It is one of the bigger real-world applications that is registered in the official HASKELL pages.[10] It is worth mentioning that TRUTH is one of the few tools listed there which was developed *using* but not *for* functional programming.

## 2.3 Characteristics of verification tools in general

The peculiarities of the verification tool domain from the point of view of software engineering were considered by Stevens in [9]. Here we briefly summarise and then focus on the implications for language choice.

Verification tools answer precisely defined questions about precisely defined systems. Thus it is comparatively easy to understand what it means for the tool's behaviour to be correct. The downside is that certain

[5] `http://www-i2.informatik.rwth-aachen.de/Research/Truth/`
[6] `http://www.haskell.org/ghc/`
[7] `http://haskell.cs.yale.edu/happy/`
[8] `http://www.informatik.uni-bremen.de/daVinci/`
[9] `http://www.graphviz.org/`
[10] `http://www.haskell.org/practice.html`

classes of bugs are unacceptable in a verification tool; semantic correctness is vital. Thus any language features supporting the development of correct programs are highly desirable.

A further characteristic is that verification tools *tend* to be developed in research environments, where it is more easily recognised for novel theoretical contributions, or new applications of theory, than for the application of "best practice" in software engineering, which is likely to be discounted because it is not new. Anything that speeds up development is an advantage, as it enables the developers to spend a higher proportion of their time on the work which is most valued. In such environments, it is also difficult to justify spending large amounts of effort on academically uninteresting aspects of the tool, such as a GUI, or on "invisible" areas such as testing(!), documentation, and ensuring portability. Nevertheless, the usability and, ultimately, success of the tool depend heavily on such aspects. Therefore, those planning to develop verification tools will do well to choose a language in which professional results can be achieved with a minimum of effort.

It is perhaps instructive to note that in some cases, the same considerations may apply to those developing languages and their associated tools.

## 3 The space of programming languages

Clearly HASKELL and SML, the languages of TRUTH and the CWB, have a great deal in common: both are basically functional languages and both have static type systems which are *strong* in the sense that a well-typed program will be free of certain classes of runtime errors. Moreover, both are minority languages, with their origins in academia. What is the significance of these features for verification tools?

**The functional paradigm.** Essentially, a functional programming language is one in which the natural programming style includes treating functions as first-class concepts. For example, one expects to write *higher-order functions*; that is, functions which take other functions as arguments. There is, however, no universally agreed definition of what it is to be a functional programming language, although no reasonable definition would exclude either SML or HASKELL. The difficulty stems from the impure nature of most languages, which stems in turn from the need to permit the use of whichever paradigm is most appropriate for a particular problem. It is possible, for example, to write higher-order functions in C; the reason why C is not included in definitions of a functional programming language is that this is not the natural, normal way to solve problems in C.

The main reason, in our view, for using a functional language for a verification tool is that the paradigm is a good match for the domain, as the most important concepts in the domain tend to be algorithms. It is often claimed that programs written in functional languages are easier to reason about, and hence are more likely to be correct, than those in one of the imperative paradigms (procedural or object oriented). The theoretical concept on which the claim rests is *referential transparency*, essentially the fact that identifiers are used for values, rather than for references whose values may change. Where this property holds, it can indeed facilitate reasoning, at least in small pieces of code. However, we have found that in practice, building a verification tool in a way which provides reasonable modularity and efficiency necessitates the use of "impure" features of the languages, so that referential transparency is lost.

Today the most obvious alternative to the functional paradigm for a verification tool writer is the object-oriented paradigm. The main argument in favour of the functional paradigm is that the most important concepts in the domain tend to be algorithms, not objects. In this respect the verification tool domain differs from most business domains, and the use of a less popular language may be justified. However, as we shall see, being out of the mainstream carries disadvantages sufficient to give one pause.

**Static typing.** In a statically typed language, the compiler carries out certain checks to ensure that the program is free of certain types of errors which might otherwise cause incorrect behaviour at runtime. This does not, of course, ensure that the program is free of errors, but it can enable errors to be caught early and easily corrected, thus speeding up the development process. Static typing is often criticised for being inflexible; but when such criticisms are investigated, they turn out to be criticisms of the inflexibility of a particular type system. We will give examples of such inflexibilities in Sect. 4. We argue that a coherent understanding of a solution to a problem includes an understanding of the types of the entities involved; if these fit the type system of the language concerned, it is hard to see how having errors caught by the compiler can fail to be a benefit, although one could still argue about the size of the benefit.

It is clear from the successes achieved by certain groups working with dynamically typed languages such as Erlang (in the functional world) and Smalltalk (in the object-oriented world) that it is possible to write complex, correct software without static typing. However, none of the authors would willingly give up the benefits of static typing. We will discuss particular features of the type systems of SML and HASKELL below.

## 3.1 Standard ML

Standard ML ([6]) is an essentially functional language in the sense discussed above. By "essentially" we mean that it is not a *pure* functional language: for example, references are permitted. ML originated at Edinburgh in the late 1970s. Research and experimentation continued over the succeeding decades in several centres, spawning a family of ML-like languages. In what is commonly regarded as the mainstream, a formal language definition was produced; this defined "Standard ML", or SML90. From early on, there were several reasonably faithful implementations of this standard. Later a major rewrite of the original language definition resulted in the new definition of Standard ML, sometimes referred to as SML97. Other notable ML-like languages are Caml and O'Caml. Although they have enough similarities to SML that many of the same considerations will apply, there are also some significant differences which might affect a user's choice. In particular O'Caml's support for software architecture is radically different, incorporating aspects of object orientation. We do not consider these languages in this paper, since our experience is with SML.

Technically, the revision to SML97 has been a substantial improvement; but it has led, temporarily at least, to difficulties of tools and libraries not all being updated at once; old SML programs cannot be compiled by new compilers and vice versa.

A variety of compilers is still available for Standard ML; by far the most widely used is Standard ML of New Jersey (SML/NJ), and this is the only compiler supported by the CWB.

The definition of Standard ML includes the Standard Basis Library, providing such things as string manipulation, operating system interfaces, and basic data structures. SML/NJ comes with a more extensive library.

## 3.2 Haskell

HASKELL is a purely functional programming language [8]. The current standard is HASKELL98, which fixes the syntax and semantics as well as a large set of standard libraries.

Until recently the embedding of input and output operations, which have to be considered as side effects, in purely functional programming languages was generally poor. *Monadic I/O* is a very elegant approach to overcoming this problem [11]. HASKELL supports this concept and supplies versatile I/O libraries offering exception handling and file manipulation operations, which were of great help in building a user-friendly and reliable tool.

## 4 Comparison of language design features

We begin by considering and comparing the more technical aspects of SML and Haskell, before going on to consider non-technical questions in the next section.

### 4.1 Typing

As semantic correctness is crucial to any verification tool, it is natural to believe that a strong static type system, enabling a large class of errors to be caught at compile time, is a good thing in a language for verification tools. Our experience supports this; although verification tools have been written in Lisp, for example, we would not like to give up the static typing provided by both SML and Haskell. The type systems of SML and Haskell are actually rather similar. In this subsection we begin our discussion by considering two related features which Haskell and SML have in common: parametric polymorphism and type inference. In the following subsections, we shall discuss the major differences between the languages' type systems separately.

Extensive type inference is convenient especially in functional programming where identifiers often have complex higher-order types. However, it has serious drawbacks for maintainability of code. The human reader of code needs to understand the types involved, and it is frustrating to know that the compiler has worked out information which is not available to the reader. The natural response is that good programming practice is then to include type annotations; but we have found this hard to put into practice. An annoyance is that the syntax of Haskell sometimes makes this impossible. For example, in Haskell function types are implicitly all-quantified and thus it was not possible to give type annotations for certain local functions. This has been remedied with so-called *scoped type variables*, which have been introduced in GHC around version 4.03 (too late for Truth), but are *not* legal Haskell98.

A more serious point which applies even to SML which does permit type annotations is that if type annotations are included which are descriptive enough to be helpful, they are too specific to allow reuse through parametric polymorphism. On the other hand the most general type is – except for utility functions – often meaningless to the programmer and fails to document the true intention of the function. For example, perhaps the programmer writes a function whose first argument has most general type $\alpha\ \mathsf{list} \times \beta$. Maybe there is initially only one application of this function, to an argument of type $\mathsf{action\ list} \times \mathsf{string}$. It may be that the fact that the first argument has type $\mathsf{action\ list}$ is essential to the nature of the function; for example, perhaps this is reflected in the name of the function, and using the function on any other kind of list would be confusing. However, perhaps the type of the second argument is less important, and if the code works on another type, the programmer may be quite happy to see it used on that type. "Morally", the function's argument has type action list $\times\ \alpha$. If the programmer thinks this through, it is possible to annotate the function accordingly; but it is not natural to do so, since it involves thinking about all possible future reuse of the code at a time when it is more appropriate to concentrate on the initial intended use. One could argue that this is what comments are for, but the advantage of type annotations is that the compiler can *automatically* check that they are consistent with actual code.

There is a tension between trying to enable code reuse on the one hand and on the other hand trying to make code understandable and trying to maintain appropriate encapsulation barriers. We find that these last two, though different, often go together: one encapsulates the definition of an important type together with appropriate functions for manipulating it, and then uses the new type name in type annotations to elucidate the code. However, in doing so one loses the power of parametric polymorphism for code reuse in clients of this new type because clients cannot see the structure of the type.

For example, processes in the process calculi we work with can have *restrictions* applied to them. A restriction is conveniently implemented as a list of actions, but certain invariants need to be maintained. If we allow clients to see that a restriction is a list of actions, then when they manipulate processes they can use the standard list functions on the restriction, but we cannot easily enforce the invariants. On the other hand, if we use encapsulation to make available only a type *restriction* so that we can enforce the invariants, we have to provide all necessary functions for manipulating this type. This is not unreasonable: it is the same work, for example, that we would have to do if we worked in an object-oriented language and created a class *Restriction*. However, when we have a variety of slightly different kinds of restriction, we have to implement the manipulating functions afresh every time; to gain encapsulation we have lost parametric polymorphism as a reuse mechanism, and we do not have inheritance available to us as an alternative mechanism. This kind of situation arises very frequently in both the CWB and TRUTH because we write code to deal with variants of process algebras and logics and with variously processed versions of them.

An additional issue in SML is that it is sometimes difficult to decide whether a conceptual type should be implemented at the module level or only at the core level; in the CWB we generally resolve this by using both but not revealing that decision outside the module where it is made, so that, for example, the signature for processes exports only a type *restriction*, whereas an implementation of that signature typically builds a structure *Restriction*, exporting a type from the content of that structure. (Note that because of the divide between module and core level, we do not have the option of working *only* at the module level; in order to write functions that work with restrictions – which is essential – at some point we have to decide what type a restriction has. So functor construction and multiple applications of functors do not solve this problem, though they may contribute to a solution.)

Further, we find that the powerful type systems of SML and HASKELL are a mixed blessing, often leading to complex type errors which are understandable only to people who are familiar with the subtleties of the respective type system. This is to some extent inevitable in a language whose standard idioms involve complex higher-order types, but refinements such as SML's equality types and weak types add to the problem, since the need for these is not easy for the non-type-theorist programmer to understand. Recent work on more informative error messages, such as [4], is to be welcomed, but has yet to make a difference to the compilers. Furthermore, an interactive *type analyser* would be desirable, a tool which, requested by the user, would visualise the types of certain subexpressions. In the meantime, our advice is that there is little to choose between SML and HASKELL in this respect.

## 4.2 Strictness vs. laziness

The most obvious difference between SML and HASKELL is that SML is *strict* whilst HASKELL is *lazy*. For discussion of the concepts in general see, for example, [10]. Basically, laziness means that values are only computed on demand, allowing the implementation of infinite data structures. In contrast, strictness refers to the fact that e.g. the arguments of a function call have to be evaluated before executing the call, no matter whether they are required or not.

In the context of verification tools, laziness seems to be an appealing feature because one might hope to get "for free" certain "on-the-fly" verification techniques that normally have to be worked out in each special case. For example, consider a class of verification questions concerning a system, such as the model-checking problem "does this system satisfy this property". To answer some questions in the class, it will be necessary to calculate the entire state space of the system. For others, only a small part of the state space, perhaps that reachable within three transitions from the starting state, will be relevant. A *global* algorithm is one which always calculates the whole state space; a *local* one does not. Local algorithms are generally harder to design and verify than global ones and often have poorer worst-case complexity, though in practice they may perform much better. One might hope to be able to get a local algorithm from a global one "for free" using laziness because the code for calculating certain parts of the state space would simply never be evaluated if its results were not called for. In practice, however, the TRUTH team found that one has to implement the algorithm generating the state space carefully in order to guarantee the desired behaviour. For example, the use of monads (cf. Sect. 4.3) or of accumulator techniques can easily destroy the on-the-fly property. Since there are no visual clues (program annotations) in the source code, it is often not entirely obvious why a function is not as lazy as one would have hoped when writing the code. Also, code which involves exceptions or destructive updates of data structures needs to be crafted quite carefully in a lazy context. Eager evaluation, on the other hand, is easier to write, comprehend, and debug because things happen deterministically, in the order dictated by the source code.

Altogether the effort required to preserve the locality of a lazily evaluated, global algorithm often corresponds to the design of an algorithm which is local by nature.

Summarising, lazy evaluation is an attractive feature, but the TRUTH team would have liked a flexible mechanism with which to specify parts of the program, which should be evaluated eagerly or lazily.

## 4.3 Imperative features

Both the TRUTH and the CWB team have found imperative features to be essential. Sometimes the concern is efficiency, but more often it is understandability: many of the algorithms we wish to implement are conceived imperatively, and in such cases implementing them functionally makes the implementation more difficult to read and hence more likely to contain errors. Prominent examples of algorithms with an imperative character are graph algorithms, which play an important rôle in tools such as ours which deal with transition systems. The data structures we deal with grow too big to keep several copies in memory, and the usual way to extract information from them is to walk them in a given order, collecting information and destructively updating the structure on the way, which can be straightforwardly described and efficiently implemented in imperative ways.

Here SML scores by providing imperative features in the core language in a reasonable and powerful way, although they can be syntactically awkward. I/O is supported by the Standard Basis Library. HASKELL uses *monads* for destructive updates and I/O; they add a restricted form of stateful computation to a pure language, retaining referential transparency ([11]). The disadvantage is that programs become more complicated. Also, if imperative elements of a given application were not taken into account during its design but turn out to be necessary later on, often major parts have to be redesigned or (at least) reimplemented, especially because types change significantly. A simple but recurring example is to add printing of status information to an otherwise purely functional algorithm. In the worst case this could result in having to rewrite the algorithm in a monadic style, but also to rewrite its callers (and transitively their callers as well), plus adjusting all type annotations on the way. Even when using opaque accessors to data structures, the required changes cannot necessarily be limited to a single module, but affect large parts of the system. This is clearly undesirable from a software engineering or economical point of view. Indeed for certain parts of the TRUTH system a redesign turned out to be necessary in the past, mostly in order to implement more efficient versions of algorithms by introducing imperative constructs like destructive updates.

The TRUTH team considers this point as one of the biggest drawbacks of the purely functional paradigm as followed by HASKELL.

## 4.4 Architecture support

The architecture of a system makes a vital contribution to its correctness. We hope to study module systems in this context, building on [7], in future; in this paper we can only indicate the main issues.

A HASKELL *module* defines a collection of values, datatypes, type synonyms, classes, etc., as well as their import/export relationship with other modules. Overloaded functions are provided in a structured way in the form of *type classes*, which can be thought of as families of types (or more generally as families of tuples of types) whose elements are called *instances* of the class. In the instantiation the definitions of the overloaded operations are given.

In Standard ML, *structures* provide the main namespace management mechanism; they may contain substructures, functions, values, types, etc. A structure may be coded directly or produced by the application of a *functor*, which may be thought of as a generic or parameterised structure. The programmer may define *signatures* which act as the types for structures; for example, a functor may be defined to take, as argument, any structure matching a given signature. The module system is separate from the core language; one cannot, for example, apply a functor conditionally. Whilst this keeps the language definition clean, in the CWB it has often caused problems leading to code duplication. The changes made to SML in SML97 are welcome; the elimination of structure sharing and the introduction of "where" clauses have solved several long-standing problems for the CWB.

A basic facility which is desirable in a module system is that it should be possible to define an interface to a module separately from the module itself. This helps developers to understand the system, as they can read interfaces to modules without being distracted by implementation information. We also want to be able to apply the same interface to several modules and to provide several interfaces to the same module. In both the CWB and TRUTH this need arises, for example, because we often work with several variants of a process algebra, logic, or algorithm which share an interface. We want the compiler to do the work of making sure that the modules stay consistent, and we want to avoid duplicating code. SML's signatures support this way of working reasonably well, although not without problems. The TRUTH team has found that HASKELL does not support this situation so well: inside the module export list, entities cannot be annotated with types, so a common practice is to add them in comments. However, this is error prone, since there is no way for the compiler to enforce their correctness or check their consistency with the implementation in the module body.

We feel that SML's architectural features are better suited than HASKELL's to our purposes; neither is ideal, however, and this seems an interesting area for future study, especially as we do not think that the class and package systems of C++ or Java would be ideal either.

## 4.5 Exceptions

The CWB used to make heavy use of exceptions as a control flow mechanism. This led to correctness problems because the compiler could not check whether or not exceptions were always handled. A common class of bugs occurred when a programmer added a new piece of functionality to the CWB by adding a new module declaring an exception; the exception could arise outside the new module; but the programmer did not, for whatever reason, modify the CWB's top level module to handle the exception sensibly. To make matters worse, in SML90, although one could write a handler that would catch all exceptions (using a wildcard), so that at least the user would not see the CWB "crash" in the case of such a bug, one could not tell dynamically which exception was actually being handled. This would result in a message to the user of the CWB along the lines of "Sorry, an ML exception has been raised. This is a bug: please report it." The exception mechanism has been improved in SML97 compared with SML90: it is now possible to interrogate an exception for its identity, which at least enables the CWB to give a fuller error message, which is useful for debugging.

Still, in a language with type safety as a strength, it is a pity to use a programming style in which the programmer cannot be certain that all exceptions are handled. From the point of view of the user of a verification tool, it is not very much better for the application to terminate because of an unhandled exception than it would have been for it to terminate because of a runtime type error. Therefore, the CWB now uses exceptions in a more disciplined way which seems to work well. A small number of specified exceptions (corresponding to such things as "error in user input", "assertion violated", etc.) are allowed to rise to the top level and are individually handled there. All other exceptions are kept within small pieces of code (e.g. within one SML module) and in each case the programmer verifies by eye that the exception cannot escape. Because the latter is hard work, exceptions are only used where the alternative is really painful.

We are not claiming, of course, that SML compilers could and should check whether exceptions are always handled; this has been a research topic for some time. Java's requirement that functions document (certain kinds of) exceptions that may be raised is an attempt to address the problem, but it is clumsy and interacts badly with a functional programming style. Our point is that the style of Standard ML programming often seen and encouraged, relying heavily on exceptions, has serious disadvantages which the software engineer needs to guard against.

The Truth team found that exceptions interacted badly with laziness: as a rather disturbing effect, partial evaluation enables exceptions to escape from an enclosing exception handler. To get the exceptions actually raised inside the handler, initially the Truth team had to resort to code like `if x==x then x else x` to enforce the evaluation of `x` at the right time. Recently, better ways to trigger full evaluation have been provided (`deepSeq $!  x`), but they are non-standard and of course destroy laziness. We think it would be much more natural to avoid situations of this kind by adopting strict rather than lazy evaluation as the standard strategy in the language. Laziness, which is a very costly feature, could then be provided upon request, using annotations of the function and constructor symbols.

We have often seen programming languages compared on the basis of how many lines of code it takes to implement some piece of functionality. We consider this a poor metric. The length of a piece of code is not well correlated either with the time it takes to write it or with the time it takes to understand it; a short piece of code may well be harder to write and to maintain than a longer one. This is why we have not tried to compare SML and Haskell on this point.

## 5 Comparison of non-language design features

### 5.1 The available compilers and their characteristics

There are now three main freely available SML97 compilers, SML/NJ, Poly/ML, and Moscow ML. (Harlequin MLWorks ceased to be available when Harlequin was bought: there was hope that it might become open source, but this now appears unlikely.) SML/NJ can now produce native code for many platforms, which is important for a widely distributed verification tool. However, it needs a third-party utility to produce stand-alone applications, and even then there is a problem with running the application from outside its directory. This is hard to explain to users of the CWB and causes embarrassment.

For Haskell, too, three compilers are available, all of them freely: NHC98, HBC, and GHC. For Truth, only GHC was considered feature-complete enough; it also provides some extensions to the HASKELL98 language which have proven helpful, for example multiparameter classes and existential types.

## 5.2 Libraries and associated tools

Good libraries and tools can help to ensure correctness (e.g. because well-used libraries have been debugged by others) and can cut down development time. We consider and compare what is available for HASKELL and for SML.

**General-purpose libraries.** Both HASKELL and SML come along with standard libraries specified alongside the language. SML97 defines the Standard Basis Library[11] (ML97SBL); HASKELL98's libraries are described in the Library Report[12] (H98LR). Broadly similar, these provide basic data structures, interface to the operating system, etc. Both GHC and SML/NJ ship with some extra, non-standard libraries.

**GUI libraries.** There is an X Window System toolkit, eXene,[13] written in Concurrent ML, though for a long time this was apparently not usable with SML97 (because of a signal-handling bug, fixed more recently than the last major CWB changes). Research projects have provided portable GUI library facilities for use with Standard ML, such as sml_tk.[14] Thus one can implement a GUI in SML; but really good high-level toolkits are still lacking. The CWB has made no serious attempt to do this. For HASKELL, too, some bindings for common GUI toolkits are available, but at the time GUI support was added to TRUTH none of them was regarded as stable or feature-complete enough to be usable for what was planned. In the end, the process simulation GUI for TRUTH was written in Java and was interfaced to the HASKELL part via Unix pipes. (The CWB followed a similar path in a student project, as yet unreleased.)

**Associated tools.** A debugger is invaluable in program development, especially when experimenting with verification algorithms which may contain bugs. Unfortunately, writing debuggers for functional languages turned out to be harder than for imperative languages like C. This is even more true for a lazily evaluated language like HASKELL, where the inspection of a value would sometimes change the evaluation order. Nevertheless some attempts have been made in this direction, mostly resulting in so-called *tracers* (like Freya, Hood, or Hat), which can record program runs for later analysis.

[11] `http://www.smlnj.org/doc/basis/`
[12] `http://www.haskell.org/definition/`
[13] `http://people.cs.uchicago.edu/~jhr/eXene/`
[14] `http://www.informatik.uni-bremen.de/~cxl/sml_tk/`

None of them was used during Truth development because they were either not available at that time or did not support some of the GHC features used in Truth. There is no debugger available for SML/NJ. There is, however, a debugger for Poly/ML, which is a welcome development. We have not yet used it.

There is a lexer (ML-Lex) and a parser generator (ML-Yacc) for SML. These were long unavailable in SML97 versions but do now seem to work (see below re documentation). The CWB uses ML-Lex but does not use ML-Yacc. At a very early stage a hand-built parser was produced, and by the time the major reengineering work was done on the CWB its syntax (perhaps unfortunately, but understandably) included features which were not supported by ML-Yacc, so that to move to ML-Yacc at that point would have involved a user-visible syntax change. This is an example of the problems which can arise when a suitable third-party component is not available at the right moment; users may not have the option of adopting it later. As mentioned, Truth uses the Happy parser generator.

Overall there is little to choose between Haskell and SML in this category, but both suffer from being minority languages. There are few providers of libraries and tools, and key developers are often more concerned with compilers. This is understandable, but to us libraries and tools are just as important.

## 5.3 Documentation and other sources of help

Famously, Standard ML has a formal specification [6], but this is impenetrable to most programmers. Fortunately there are also several accessible books and tutorials available. The official specification of Haskell is given by the Haskell98 Language Report,[15] which defines the syntax of Haskell programs and gives an *informal* abstract semantics. For such a technical document it contains much plain text, and the general impression of local Haskell developers is that it is quite readable. On the other hand, as was noted elsewhere:[16] "The informal specification in the Haskell report leaves too much room for confusion and misinterpretation. This leads to genuine discrepancies between implementations, as many subscribers to the Haskell mailing list will have seen."

Regarding the compiler and associated tool documentation, the overall impression of the authors is that GHC's documentation is slightly better than that of SML/NJ. (The ML-Lex documentation has not been updated for SML97, for example.) This has not always been the case, but the GHC developers have improved the documentation quite a lot in the recent past.

[15] http://www.haskell.org/onlinereport/
[16] http://www.cse.ogi.edu/~mpj/thih/

In both cases documentation for libraries is patchy, especially in the case of compiler-specific libraries, where it sometimes happens that the programmer must consult the source code to get more information than the signature of a function. H98LR and ML97SBL are better documented. From 1997 to 2001 there was no complete and up-to-date documentation of the latter, which was a serious problem. Now, however, an updated web page is available;[17] the documentation appeared in book form [2] in 2002, which is a welcome development. The CWB and TRUTH teams each had the impression initially that the other's language's libraries were better documented: this may reflect that one notices faults only on close acquaintance. A plus for HASKELL is that the GHC library documentation has a consistent history of being frequently updated and improved.

Moving from documents to people as sources of help, we have found the newsgroups `comp.lang.ml` and `comp.lang.functional` and the GHC mailing lists to be useful. Naturally it is easier to get help with problems which can be described briefly. When we have needed help with, for example, making architectural decisions, local language experts have been invaluable; this is something that developers should bear in mind.

Last but not least the home pages of Standard ML of New Jersey[18] and of HASKELL[19] provide useful collections of links and references to other resources.

## 5.4 Foreign-function interfaces

We have made no serious attempt to bind C and SML code within the CWB, because the Standard ML foreign-function interfaces were perceived (and experienced in a student project) as hard to use and inefficient. Matthias Blume's new "NLFFI" foreign-function interface may well change the situation.

The foreign-function interface in HASKELL has undergone a major redesign and is now quite usable. TRUTH has been extended by a parallel model-checking algorithm, which uses the FFI layer to call C functions from the MPICH resp. LAM libraries, both well-known implementations of the *Message Passing Interface* standard. For this application the *marshalling* required to convert between HASKELL and C data formats turned out to be very inefficient, however. Another problem was the instability of the FFI interface at the time the TRUTH team were using it: it changed rapidly between releases of GHC. The TRUTH team made extraordinary use of preprocessor directives and *autoconf* magic in an attempt to allow TRUTH to support many compiler versions, but

[17] Unfortunately, it documents the version of the library corresponding to the very latest "working", i.e. experimental, version of the compiler. There still seems to be no freely available documentation for the version of the library in the latest release-quality version of the compiler!
[18] `http://www.smlnj.org/`
[19] `http://www.haskell.org/`

they were eventually forced to give up.

## 5.5 Stability of languages and their implementations

The current stable version of HASKELL is HASKELL98, dating from 1997. This is the fifth major version of the language definition (the next will be *Haskell 2*!). Compilers, of course, provide the effective definition of the language. There have been many changes in what GHC supports (e.g. multiparameter classes, implicit parameters). Not all changes to extensions have been backwards compatible, which is inconvenient for programmers who need those extensions.

Regarding the stability of HASKELL *implementations*, only GHC has been thoroughly examined, since the other implementations have been ruled out by other issues, as stated earlier. GHC is under steady development, and the quality of released versions differs greatly. Some are quite stable, but for others patch-level releases have to be made quickly to fix the worst bugs. Unsurprisingly, bugs often accompany new features. In fairness, bugs are fixed promptly by the GHC developers once they are reported to the relevant mailing list. However, faced with a show stopper, an application programmer must choose between waiting for an official release of GHC including a fix, or becoming expert in building the compiler itself, which is non-trivial, time consuming, and will not further his/her aims.

Standard ML was subject to a major revision, from SML90 to SML97. The SML/NJ compiler has undergone many releases, but now seems fairly stable. As mentioned, tools and libraries tend to lag. The ML2000 project intended to develop a future version of ML. Little has been heard of the project recently, and many of its early ideas have been incorporated in O'Caml. We believe that SML97 will increase, rather than decrease, in stability over the next few years.

## 5.6 Performance

This is a controversial topic, but it is an important one for developers of verification tools. Both speed and space usage are important, with space usage often being more important, as the amount of memory used by a verification tool is normally the limiting factor. Our experience with both SML and HASKELL suggests that performance is particularly poor with respect to memory usage. Also, as noted, the key feature of HASKELL, lazy evaluation, comes at a high cost.

# 6 Conclusion

We proceed by summarising the features we found (not) helpful in using functional languages in general, and specifically in the use of SML and HASKELL.

## 6.1 Helpful features

Both teams found the functional paradigm a good fit for the domain of verification tools. The automatic memory management of SML and HASKELL lets us focus on the important parts of the algorithms instead of fiddling with implementation details. Data structures that are readily available like lists and their natural use were appreciated as well. Also, static types proved to be helpful in catching certain classes of errors early.

However, some of these features turned out to be double-edged swords, and other features which were initially considered useful turned out not to be, or to be less helpful than we had hoped.

## 6.2 Mythical silver bullets

The price of using convenient constructs like higher-order functions and functional languages in general is often paid in uncompetitive runtime and memory performance. While in imperative languages it is possible, after getting programs right, to get them *fast*, we find this harder in functional languages.

It is commonly argued that referential transparency makes programs easier to understand, easier to reason about, and generally more robust. However, we did not find this feature to be worth its cost. Constraining (in HASKELL) the use of e.g. destructive updates *when they are needed* turned out to waste a lot of precious developer time. Reasoning about any sufficiently complex algorithm *on the source code level* is intractable as well, even when avoiding impure features.

Static types were helpful, as mentioned above, but also got in our way. Understanding complex type errors and their origins requires an intimate understanding of the type system. Easier means of exploring them and deriving complex type annotations are lacking.

## 6.3 SML vs. Haskell

The most outstanding difference between HASKELL and SML is the evaluation order. While HASKELL's *lazy* approach might have its merits in some cases, it was more a hindrance than a help, causing longer development, having negative impact on the performance, and making the source code harder to understand.

While the use of *monads* in HASKELL is a nice theoretical concept to capture side effects and stateful computation in a purely functional setting, their disadvantages lead us to vote for the *impure* SML here, especially given that many of our domain's algorithms are naturally described imperatively.

We found the module systems in both SML and HASKELL lacking features from a software engineering point of view. For example, in SML the "include" mechanism for signatures is not flexible enough to prevent one from having to duplicate information across several SML signatures when different views onto a structure are required (comparable to "private" and "public" interfaces in languages such as Java). The lack of conditional application of functors, similarly, can lead to code duplication, which is in turn a maintenance problem. Since we have not surveyed other programming languages in this respect, it is not clear whether they would meet our demands, though.

Both teams would have wished for better support for exceptions in order to enhance runtime error messages (SML) and to better cope with astonishing evaluation order interactions (HASKELL).

Confronted with the poor availability of ready-to-use libraries for SML and HASKELL compared to mainstream languages, one is often referred to their *foreign-function interfaces*. In HASKELL, the use comes with a performance penalty. We do not have first-hand experience for SML.

Aside from the language issues we have investigated, we found the tool support in both languages lacking in comparison to mainstream languages, resulting in the reinvention of wheels other languages can readily use. We regard this an important factor when planning the development schedule of SML or HASKELL programs.

We found documentation for HASKELL to be better, while stability of the SML implementation was found to be superior when compared to its HASKELL counterpart.

## 6.4 Considerations for future projects

There have been positive and negative aspects to both our sets of experiences with HASKELL and SML, as there would doubtless have been with whatever language we had chosen. Overall, we consider Standard ML to be a slightly better choice for our kind of application than HASKELL, more because of a more stable environment of supporting tools than because of language features. Of course, there are many alternatives including other functional languages with which we have less experience; O'Caml might be a strong candidate.

However, it turned out in our discussions that none of us were enthusiastic about the idea of using a functional language for a future verification tool because of their impoverished environments compared with mainstream programming languages. Our impression was that SML and HASKELL can play out their advantages mainly in the prototyping stages of a project, an arena where both would have to compete with dynamic languages like Lisp or Smalltalk, or scripting languages like Python (which have faster turn-around cycles due to absence of a compilation phase).

Our conclusion is that, if/when we develop new verification tools, we would like to conduct a study on the uses of imperative languages for verification tools. During our investigations we got the impression that those seem to be better equipped for features we need in our domain.

We hope that reporting our experience using major functional languages will help the community to improve such languages and their worlds in future.

**Acknowledgement.** We thank the anonymous referees for helpful comments and discussions.

## References

[1] Clarke EM, Wing JM (1996) Formal methods: state of the art and future directions. ACM Comput Surv 28(4):626–643

[2] Gansner ER, Reppy JH (2002) The Standard ML Basis Library. Cambridge University Press, Cambridge, UK

[3] Hartel PH et al. (1996) Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. J Function Programm 6(4):621–655

[4] McAdam B (2002) Repairing type errors in functional programs. PhD thesis, Division of Informatics, University of Edinburgh

[5] Milner R (1989) Communication and concurrency. International Series in Computer Science. Prentice-Hall, Upper Saddle River, NJ

[6] Milner R, Tofte M, Harper R, MacQueen D (1997) The definition of Standard ML (revised). MIT Press, Cambridge, MA

[7] Nicklisch J, Peyton Jones SL (1996) An exploration of modular programs. In: Glasgow workshop on functional programming, July 1996

[8] Peterson J, Hammond K, et al (1996) Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, New Yaven, CT, May 1996

[9] Stevens P (1999) A verification tool developer's vade mecum. Int J Softw Tools Technol Transfer 2(2):89–94

[10] Wadler P (1996) Lazy versus strict. ACM Comput Surv 28(2):318–320

[11] Wadler P (1997) How to declare an imperative. ACM Comput Surv 29(3):240–263