Functional programming languages for verification tools: experiences with ML and Haskell

Martin Leucker[1], Thomas Noll[1], Perdita Stevens[⋆2], and Michael Weber[⋆⋆1]

[1] Lehrstuhl für Informatik II, RWTH Aachen
Email: {leucker,noll,weber}@informatik.rwth-aachen.de
[2] Division of Informatics, University of Edinburgh
Email: Perdita.Stevens@dcs.ed.ac.uk

Abstract

We compare Haskell with ML as programming languages for verification tools, based on our experience developing TRUTH in Haskell and the Edinburgh Concurrency Workbench (CWB) in ML. We discuss not only technical language features but also the "worlds" of the languages, for example, the availability of tools and libraries.

1 INTRODUCTION

It is surprising, and in our view unfortunate, that there is little literature comparing different functional languages. The Pseudoknot benchmark [H$^+$96] compares implementations of functional programming languages (Haskell and ML among them) with respect to their runtime and memory performance, and [NP96] compares the module systems of Haskell and ML. We did not find good sources of help for a developer trying to choose between the languages based on all relevant aspects. 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 on the basis of the language most used in their institution. 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.

Part of the reason for this hole in the literature may be that in order to compare languages it is necessary to compare them in a particular domain, and to make a serious comparison one needs comparable, non-toy applications written in each language. 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 ML and Haskell for two broadly comparable applications in the domain of verification tools: the Edinburgh Concurrency Workbench, in ML, and TRUTH, in Haskell.

⋆supported by an EPSRC Advanced Research Fellowship
⋆⋆supported by the *Graduiertenkolleg "Software für Kommunikationssysteme"*

2 VERIFICATION TOOLS

The domain of *verification tools* is eminently suited to the use of a statically typed functional language such as ML and Haskell, and both languages are popular choices. 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 ML

Work on the CWB[1] began in 1986. Core features, present since the beginning, are that it allows users to define processes in several variants on Milner's CCS (Calculus of Communicating Processes) and to manipulate and analyse these processes. For example, users may explore the behaviour of a process using the simulation command, or compare two processes using one of many equivalences and preorders, or check whether a process satisfies a formula in a temporal logic.

One major focus of the CWB was always research; it was a platform which Edinburgh researchers 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 25kloc in ML, 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 SML'90 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[2] in future. This paper, however, inevitably considers SML/NJ more than any other ML compiler.

2.2 The verification platform TRUTH, in Haskell

In terms of features, TRUTH[3] is very similar to the CWB. It also uses CCS to define processes and a temporal logic to specify properties, which then can be automatically tested using different model checking algorithms. It uses visualisation

[1] `http://www.dcs.ed.ac.uk/home/cwb/`
[2] `http://www.polyml.org/`
[3] `http://www-i2.informatik.rwth-aachen.de/Forschung/MCS/Truth/`

tools such as DAVINCI[4] and the GraphViz package[5] to help the user understand TRUTH's answers. TRUTH also employs the HAPPY[6] parser generator and many of the available HASKELL libraries, as well as JAVA and C libraries to provide functionality such as textual and graphical user interfaces and network communication. It is one of the bigger real-world applications registered in the official HASKELL pages[7]. TRUTH is one of the few tools developed *using* but not *for* Haskell.

TRUTH's initial version was developed as a master thesis project in 1997. Its original modular architecture has proved quite easy to understand, so no major restructuring has been needed so far. TRUTH now consists of approximately 18kloc in HASKELL. Despite the availability of several Haskell compilers, TRUTH is written for the Glasgow Haskell Compiler[8] (GHC), and since it uses some non-standard Haskell extensions and libraries only present in the GHC, to date no effort has been made to port it to other Haskell systems.

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 [Ste99]. Here we briefly summarise, and then focus on the implications for language choice.

Verification tools answer precise, well-defined questions about precisely defined systems. Compared with business systems, it is comparatively easy to understand what it means for a verification tool's behaviour to be correct. The downside is that certain 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 easier to be 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 depends 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 interesting and perhaps instructive to note that in some cases, the same considerations may apply to those developing languages and their associated tools.

[4] `http://www.daVinci-Presenter.de/`
[5] `http://www.research.att.com/sw/tools/graphviz/`
[6] `http://haskell.cs.yale.edu/happy/`
[7] `http://www.haskell.org/practice.html`
[8] `http://www.haskell.org/ghc/`

3 COMPARISON OF LANGUAGE DESIGN FEATURES

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

3.1 Typing

Since semantic correctness is a crucial property of any verification tool, it is natural to believe that a strong static type system, which enables a large class of errors to be eliminated at compile time, is a good thing in a language for verification tools. Our experience with the CWB and TRUTH supports this; although many verification tools have been written in Lisp, for example, we would not like to give up the static typing provided by both ML and Haskell. To go further, let us consider two related features which Haskell and ML have in common, namely parametric polymorphism and type inference.

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 the languages and the presence of identifiers with complex types sometimes makes this awkward (ML) or impossible (Haskell). A more serious point is that if type annotations are included which are descriptive enough to be helpful, their presence hinders reuse by parametric polymorphism. 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. The last two are not necessarily inseparable, but we have found them to go together often: 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 help make the code understandable. However, in doing so one loses the power of parametric polymorphism for code reuse in clients of this new type, because the 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 ML 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.

Further, we find that the powerful type systems of ML and Haskell are a mixed blessing, often leading to complicated and hard-to-understand type errors. Work on more informative error messages is to be welcomed. In the meantime, our advice would be that there is little to choose between ML and Haskell in this respect.

3.2 Strictness/laziness

The most obvious difference between ML and Haskell is that ML is *strict* whilst Haskell is *lazy*. For discussion of the concepts in general see for example [Wad96].

In the context of verification tools, laziness is 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 questions "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. Section 3.3) or of accumulator techniques can easily destroy the on-the-fly property. Altogether the effort which is required to preserve the locality of a lazily evaluated, global algorithm corresponds to the design of an algorithm which is local by nature.

3.3 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 to implement 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.

Here ML scores by providing imperative features in the core language in a reasonably and powerful way, although they can be syntactically awkward. I/O is supported by the Standard Basis Library.

Haskell's standard concept for destructive updates and I/O are *monads* which have been designed to add a restricted form of stateful computations to a pure language without losing referential transparency. The price to be paid for this is programs getting more complicated. Also, if imperative elements of a given application were not taken into account during its design but turn out to be necessary for some reason later on, usually major parts have to be redesigned or to be reimplemented at least, especially because types change significantly. 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. The TRUTH team considers this point as one of the biggest drawbacks of the purely functional paradigm as followed by HASKELL.

3.4 Architecture support

The architecture of a system makes a vital contribution to its correctness. We hope to make an in-depth study of module systems in the context of verification tools and their architectures, building on [NP96], in future; in this short paper we can only indicate the main issues.

A Haskell *module* defines a collection of values, datatypes, type synonyms, classes, etc., and their import/output 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 either 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 an argument which can be any structure that matches a given signature. The module system is quite separate from the core language; one cannot, for example, apply a functor conditionally. Whilst this keeps the language definition clean, in the CWB this has often caused problems leading to code duplication. The changes made to ML in SML'97 are very 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. ML'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 ML'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.

3.5 Exceptions

The CWB used to make heavy use of exceptions as a control flow mechanism. This led to correctness problems because the compiler cannot check whether exceptions are always handled or not. The result was that bugs could lead to an exception rising to the top level, where nothing sensible could be done to handle it. To make matters worse in SML'90, although one could write a handler that would catch all exceptions, one could not then tell which exception was actually being handled. Therefore such bugs 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 greatly improved in SML'97 compared with SML'90: 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, using a programming style in which it is impractical for the programmer to be certain that all exceptions are handled, although common, is unfortunate in the context of a language one of whose strengths is its type safety. 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 run-time 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 ML 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.

The TRUTH team found that exceptions interacted badly with laziness: in order to get their exceptions actually raised when they wanted them they had to resort to code like `if x==x then x else x` to enforce the evaluation of `x`. We think it would be much more natural to avoid situations like this 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.

Finally we want to remark that both groups have frequently seen programming languages compared on the basis of how many lines of code it takes to implement some piece of functionality. We would like to say that 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, so it may well happen that a short piece of code is harder to write and to maintain than a longer piece of code. For this reason we have not attempted to compare ML and Haskell on this point.

4 COMPARISON OF NON-LANGUAGE DESIGN FEATURES

4.1 The available compilers and their characteristics

There are now three main freely available SML’97 compilers, SML/NJ, Poly/ML and Moscow ML. (Perhaps Harlequin MLWorks may yet become open source.) 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: HBC, NHC98 and GHC. For TRUTH, only GHC was considered feature-complete enough; it also provides some extensions to the Haskell 98 language which have proven helpful, for example multi-parameter classes and existential types.

4.2 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 ML.

General purpose libraries Both Haskell and ML have standard libraries specified alongside the language. SML'97 defines the Standard Basis Library[9] (ML97SBL); Haskell 98's libraries are defined in the Library Report[10] (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[11], written in Concurrent ML, though until recently this was apparently not usable with SML'97 (because of a signal-handling bug, recently fixed). Research projects have provided portable GUI library facilities for use with Standard ML, such as sml_tk[12]. Thus one can implement a GUI in ML; 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 there have been made some attempts in this direction, mostly resulting in so-called *tracers* (like Freya, Hood, or Hat), which can record program runs for later analysis. None of them were 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 ML. These were long unavailable in SML'97 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 that can arise when a suitable third-party component is not available at the right moment; users may not have the option of adopting them later. We have already mentioned that TRUTH uses the Happy parser generator.

[9] http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/index.html
[10] http://www.haskell.org/definition/
[11] http://cm.bell-labs.com/cm/cs/who/jhr/sml/eXene/index.html
[12] http://www.informatik.uni-bremen.de/~cxl/sml_tk/

Overall there is little to choose between Haskell and ML 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.

4.3 Documentation and other sources of help

Famously, Standard ML has a formal specification [MTHM97], 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 Haskell 98 Language Report[13] 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 was that it is quite readable. On the other hand, as was noted elsewhere[14]: "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 compilers' and associated tools' documentation, the overall impression of the authors is that GHC's documentation is slightly better than SML/NJ's. (The ML-Lex documentation has not been updated for SML'97, for example.) This has not always been the case, but the GHC developers have improved the documentation quite a lot in the recent past.

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. The H98LR and the ML97SBL are better documented, but there are errors in the documentation of the latter, which is labelled Draft and dated 1997: this may be for copyright reasons connected with a forthcoming book, but it is extremely unhelpful to the programmer. 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 is 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[15] and of Haskell[16] provide useful collections of links and references to other resources.

[13] `http://www.haskell.org/onlinereport/`

[14] `http://www.cse.ogi.edu/~mpj/thih/TypingHaskellInHaskell.html`

[15] `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

[16] `http://www.haskell.org/`

4.4 Foreign function interfaces

We have made no serious attempt to bind C and ML 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 were eventually forced to give up.

4.5 Stability of languages and their implementations

The current stable version of Haskell is *Haskell 98*, 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. multi-parameter 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 remaining implementations have been ruled out by other issues, as stated before. GHC is under steady development and the quality of released versions differs greatly. Some are quite stable, but for others patchlevel releases had to be put together shortly after to fix the worst bugs. Unsurprisingly bugs often accompany new features. In fairness, bugs are fixed very quickly by the GHC developers once they are reported to the relevant mailing list. However, the bottom line is: for real show stoppers, an application programmer can choose between waiting for another official release of GHC, which includes these fixes, or becoming an expert in building the compiler itself, which is non-trivial and time consuming, and after all not exactly part of his/her duty.

Standard ML has undergone a major revision, from ML'90 to ML'97. The SML/NJ compiler has undergone many releases, but now seems fairly stable. As has been mentioned surrounding tools and libraries tend to lag. The ML2000 project intends or 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 SML'97 will increase, rather than decrease, in stability over the next few years.

4.6 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 is normally the factor limiting what can be done.

Our experience with both ML and Haskell suggests that performance is particularly poor with respect to memory usage. Additionally, as we have noted, the key feature of Haskell, lazy evaluation, comes at a high cost.

5 CONCLUSION

There have been positive and negative aspects to both our sets of experiences with Haskell and ML, 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 of 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 conclusion is that if/when we develop new verification tools, these will be written in "a Java-ready subset of C++". That is, we would prefer to be writing in Java, but at present this appears premature for performance reasons. We would write therefore in C++, trying to avoid using features (such as multiple inheritance, non-virtual functions, "friends") which cannot be readily translated into Java.

We hope that this paper's example of real experience using major functional languages will be useful to the community in improving such languages and their worlds in future.

REFERENCES

[H^{+}96] Pieter H. Hartel et al. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[NP96] J. Nicklisch and S.L. Peyton Jones. An Exploration of Modular Programs. In *Glasgow Workshop on Functional Programming*, July 1996.

[Ste99] Perdita Stevens. A verification tool developer's vade mecum. *Software Tools for Technology Transfer*, 2(2):89–94, 1999.

[Wad96] Philip Wadler. Lazy versus strict. *ACM Computing Surveys*, 28(2):318–320, June 1996.