Type-Safety of Standard ML

Michael Ashley-Rollman
Advised by Karl Crary

April 30, 2005

Abstract

While a lot of research has gone into the study of type-safety, it has been done with simple contrived languages. There has never been a rigorous formal proof of this important property for any real language. Also, some prior work has gone into providing a definition of Standard ML [1], but it is known to contain bugs and was done entirely by hand. This could be greatly improved upon by a machine checkable definition of Standard ML along with a proof of type-safety for it. We have done this using Twelf, a formal logic tool designed for formalizing logics and formulating meta proofs.

Contents

1 Introduction

1.1 Some Definitions

1.1.1 Type-Safety

Type-safety is the property of a program of never getting stuck. This means that, when run, the program will either loop forever or return a value. This means that it can never seg-fault or get stuck in other ways. We say a language is type-safe when every program in that language is type-safe. Type-safety is typically proved using the preservation and progress lemmas. Preservation states that if a well-typed program or expression takes a step (is a evaluated a little) then the resulting expression is still well-typed. Progress states that any well-typed expression is either a value or can take another step.

1.1.2 Regularity

Some other nice features of programming languages are collectively called regularity conditions. These conditions on the language state things like if some expressions e has a type t, then t is a well-formed type.

1.2 Motivation

It has long been believed that Standard ML is a type-safe language, but there has never been a formal proof of this property. In fact, type-safety has not previously been proved for any real language. While type-safety has been much studied, it has remained a theoretical property of small contrived languages. It is time for this theoretical property to become real.

There are other problems with the current state of the art in programming language design. The current best for designing languages is to write down a series of judgements on paper and assert that they are reasonable. It is not uncommon for these definitions to contain errors and omit rules and cases. It is then necessary, when implementing a compiler or interpreter, to reason about the rules and exactly what they mean. When bugs become apparent, it is necessary to determine the "correct" way to resolve them and this can lead to different results in different implementations.

All of these issues have been addressed simultaneously by defining Standard ML and proving it type-safe in Twelf. Twelf is an implementation of LF based on the Curry-Howard isomorphism. LF is a framework for defining logics such as Standard ML. Twelf allows us to prove metatheorems about these logics. By defining a language in Twelf many possible opportunities for bugs are eliminated. Harper-Stone, for example, uses a judgement which is not defined (see section 5). Twelf does not permit this and thus eliminates this opportunity for a bug. Further bugs become apparent while proving theorems, such as regularity and type-safety, about the language as the bugs frequently make these theorems false. The theorems themselves must be correct to pass Twelf's theorem checker.

In addition to these correctness benefits, Twelf provides the unique opportunity of running the definition itself via proof search. While this is incredibly inefficient, it eliminates the need to interpret the definition to determine how corner cases should be handled and provides an accurate metric for determining the correct behavior of compiler test cases. It should be possible to improve the efficiency by using real numbers when running proof search rather than zero and successor.

2 The Elaborator

The Harper-Stone definition of Standard ML[1] provides an explicitly-typed lambda calculus referred to as the internal language or IL and an elaboration from Standard ML (called the external language or EL) to the internal language. The internal language is simpler than Standard ML as the syntactic sugar and other redundancies have been removed without losing any expressive power. As the elaborator provides a direct mapping from ML programs to IL programs, it is immediately clear that the IL is at least as expressive as the EL. Furthermore, it is sufficient to prove theorems for the IL since every EL program maps to an IL one.

Another benefit of this method is that new languages can be defined by an elaboration to the existing IL and thus immediately obtain any results shown for the IL for free. If the IL is not sufficiently rich to support an elaboration from a new language then it can be extended and Twelf will guarentee that the relavent cases will be added to all the existing theorems. This provides the opportunity of extending Standard ML and of designing new languages and obtaining safety guarentees with little effort.

At this point in time, the IL is defined in Twelf (see Appendix A), but the elaborator exists only in the Harper-Stone paper[1]. While the elaborator should be written in Twelf and could probably be so without significant difficulty, this has not yet happened. There are no interesting theorems to prove about the elaborator; it is only necessary to show that one exists. This has already been done by Harper and Stone on paper[1] and so the only benefit of doing it in Twelf would be to show that their work is correct. This will be done at a later time.

3 Methodology

3.1 Naming Scheme

When approaching a large project it is important to remain organized or else the whole thing is likely to collapse around you. One way of resolving this problem is by using module systems to break a large project into smaller pieces. Unfortunately for us Twelf does not yet have a module system so we did not have this option. Instead we attempted to simulate it as best as we could through the way code was divided among files and the general naming scheme.

When picking names for judgements, rules, lemmas, etc., we tried to maintain a convention of separating parts of the name with slashes and pretending that everything to the left of the slash was the name of a module and everything to the right was the name of the specific judgement or lemma. `ok/con`, for instance, means that `con` is a judgement in the `ok` module. Similarly `ok/kind`, `ok/exp`, and others represent judgements in the `ok` module. All of these judgements are in the same "module" because they have similar meanings - they all express the well-formedness of something. While this convention can sometimes lead to very long names for heavily nested modules, it was mostly successful in preventing name-space collisions and allowing rules names to be easily remembered. It was, however, more difficult to make lemma names follow this convention while adequately expressing the meaning of the lemma in the name.

A naming scheme for variables in judgements and lemmas was also used and was much easier to stick to. Variables were usually named for the first letter or two in their type - `E` or `e` for expressions, `C` or `c` for constructors, etc. If there were multiple variables with the same type, then they were

distinguished with primes or numbers - `E'` or `E2`. Derivations were named after the primary variable that they were related to along with a brief name in lower case saying what sort of derivation it is. Thus `Cok` would be a derivation of `ok/con ST C K` and `C'ok` would be a derivation ok `ok/con ST C' K`. If there are multiple derivations that have the same main variable then a prime is added on the end of one of the derivation names. Thus `Cok'` might be a derivation of `ok/con ST' C K`.

3.2 Other Choices

One of the nice features of Twelf is that it allows you to use the Twelf context to represent the context of the logical system being defined in Twelf. One of our first choices was to take advantage of this opportunity to eliminate the need to represent the context explicitly or prove substitution (as it follows for free from Twelf's substitution). This worked out fairly well, except for some issues involving the store. The issues that arose here are explained further in sections 4.2 and 4.3. Despite these complications, using Twelf's context was a definite win.

Another choice that we made was to faithfully follow the Harper-Stone[1] definition of Standard ML. While we considered revising the definition to use singleton kinds instead of translucent sums, we decided that it was best to use the existing definition for two reasons. Firstly we had a better idea of what the issues were that we would run into using the existing definition. Secondly our primary goal was to show that an existing language could be formalized in a computer-checkable way and important theorems like type-safety could be proved. We felt that sticking with the existing definition was more faithful to this goal. We later came to regret this decision when it became necessary to define algorithmic constructor equivalence (as mentioned in section 4.5).

Despite trying to remain faithful to Harper-Stone[1], there were a few differences. First of all we omitted signature equivalence as it does not come up anywhere in the definition and is, therefore, unnecessary. Another less significant change was the choice to omit floats. Adding floats to the language should not make the proofs any more difficult than adding integers and characters, but the process of formalizing the IEEE floating point spec is an entire project of its own. The most significant change was in the way evaluation works. Harper-Stone does evaluation using a stack, but makes no distinction between whether it is presently evaluating a term or returning a value. Instead, Harper-Stone bases its decision on how to proceed on whether the current term is a value or not. If it is a value, then it returns it. If it is not a value, then it evaluates it some more. Doing things this way would require a notion of what it means to not be a value and makes it a little more difficult to reason about what it going on. Instead, we have chosen to split the rules into those that correspond to evaluating, those that correspond to returning, and those that correspond to raising an exception. We keep track of our present state and evaluate inwards until we reach a base value (constants, empty lists, etc.) and then start returning up the stack and raising any exceptions.

Our overall approach to deciding the order in which to define things and prove theorems was to go back and forth between introducing new definitions and proving theorems about them. There were a lot of different ways that we could have interleaved making definitions and proving theorems. We could have defined everything and then proved all of the theorems or introduced a few terms from a family, proved all the theorems, and then introduced a few more terms covering all the additional cases as we went along. Our approach was to introduce entire families at a time and then prove the theorems that require them. We chose this path for a few reasons. Firstly, introducing a few terms at a time might lead to proving theorems in a way that won't work when all the remaining terms in the family have been added. It also tends to be annoying to need to go back

and constantly update theorems to cover the remaining cases. This might seem to suggest that we should have defined everything before proving any theorems, however we found that proving theorems was an excellent way to find bugs and determine what representations will and will not work. Thus, by proving theorems as soon as the families they depend on are complete, we were able to find the bugs and other issues as early as possible so that correcting them impacted only a minimal set of other things.

4 The Proofs

4.1 Regularity

Before proving type-safety, a large number of regularity conditions were proved (see Appendix B). These conditions basically tell us that anything appearing in a derivation must be well-formed. Thus, we can infer `ok/con ST C kind/t` given that we know `ok/exp ST E C` and other similar results. These proofs were done primarily to catch bugs and demonstrate that the regularity conditions in the back of Harper-Stone[1] do, in fact, hold. As it turns out, however, they were also useful during the proofs of preservation and progress.

While the proofs of regularity were fairly straight-forward, strengthening came up a few times. When one projects from a module there is a requirement that the projected expression, constructor, or module does not depend on the things that appear before it in the module. Strengthening arises when we need to find a proof that whatever is projected is well-formed. In order for the module to be well-formed, it must contain a proof that everything inside is well-formed, however these proofs may depend on the other things in the module. Thus we need a strengthening arguement to show that if the projected term is independent of the rest of the module, then the proof that it is well-formed is also independent of the rest of the module. Rather than prove strengthening, we chose to make an additional assumption that whatever was projected from the module was well-formed. This shifts the burden of strengthening onto whomever originally showed that the projection was sound.

4.2 Extending Stores

Over the course of preservation it becomes necessary to show that extensions to the store preserve typing. The basic intuition behind this is that adding new tags or references to the store doesn't break anything. Anything that was well-formed with the old store can just ignore all the new things and the old derivation is almost identical to the new one. A problem, however, arises anytime we declare variables. Consider, for example, the typing rule for a lambda expression. This rule requires `{c:con}ok/con ST c K' -> ok/con ST (C c) K` as one of the premises. This premise states that if `c` is a constructor and `c` has kind `K'` in a store with type `ST` then `(C c)` has the kind `K` in a store with type `ST`. The problem arising in evaluation is that the store can be extended over time. In particular, if there is a proof of `ok/con ST' C' K'` and a proof that `ST'` is an extension of `ST` (that is everything in `ST` is in `ST'`) then we would like to conclude `ok/con ST' (C C') K`. This does not, however, follow from Twelf's substitution as `ok/con ST' C' K'` relies on the wrong store. A substitution lemma, therefore, is necessary.

Before proving substitution, we reformulated the assumption to better indicate what was really being assumed. What one would really like to say is that if `c` is well-formed at kind `K'` in some

future store (extension) `ST'` of `ST` then `(C c)` is well-formed at `K` in that same future store. The problem is that saying that would require writing down the future store when we write down the rule. Trying to do this is futile as the same problem as before results - we don't know now what that future store will be. Instead we need to say that if `c` is well-formed at kind `K'` in all future stores, then `(C c)` is well-formed at kind `K` in the current store. Rather than write this, it is simpler to say that `c` must be well-formed at `K` in every store. This is sufficient because we know that the proof that `(C c)` is well-formed at `K` can only ever mention the current store (or if we extend the store then the extended store). We can then simplify our assumption further by just saying that `c` has kind `K'` irrespective of the store. Thus, our premise for lambda expressions is now {`c:con`} `ok/var/con c K' -> ok/con ST (C c) K`.

Once we change the rule for saying that a variable is well-formed, it is necessary to create a way to construct `ok/con ST c K'` for the premise to be useful. Originally we solved this problem by adding the rule `ok/con ST C K <- ok/var/con C K` which says that `C` is well-formed at kind `K` in store `ST` provided `C` is a variable with kind `K`. This however created problems with regularity as we need some way to show that `K` is a well-formed kind. While this is proveable since we require that `K` be well-formed before making the assumption in the first place, the proof requires adding the derivation to the context and assuming that the context contains a derivation for every such assumption. This makes things messy, so we adopted the simpler approach of modifying the variable role to be `ok/con ST C K <- ok/var/con C K <- ok/kind K`. This allows regularity to just pluck the proof that `K` is well-formed out of the derivation. Once the left side no longer mentions the store, the proof that store extension doesn't break anything goes through straight forwardly.

4.3 Substitution

Proving the substitution lemma is fairly straight forward except for an issue with dependent types. Because Twelf only provides access to the last element in the context, it is necessary to swap the order of the variables in the context when entering under a lambda as we need the variable we are substituting for to remain at the end of the context. This, however, is not necessarily possible when one variable might depend on the other. Fortunately this problem can be solved with a hack. Since there is only one variable that is being substituted for at a time, there is only one variable that could appear later in the context than it needs to. Consequently we are able to make the dependency in the backwards order by allowing the dependent variable to depend on anything provided it happens to be the right thing. Then, as the variable being substituted for is the only right thing, the proof is able to proceed.

As this method of allowing variables in the context to depend on other variables that occur later in the context is a sort of hack, the variable being substituted for is aptly marked as `hack/arg/con`, `hack/arg/exp`, or `hack/arg/mod`. Once the substitution is complete, the resulting proof may still depend on the extra assumption that the substituted term is the right thing. A number of cleanup lemmas are, therefore, necessary to remove these assumptions. As there are no constructs that are capable of relying on these assumptions, it is a simple matter of walking through the terms and demonstrating that they do not appear.

4.4 Canonical Forms

Canonical forms is a lemma which tells you what a value at a given type is. This is necessary for progress so that one can show that the expression is the right kind of expression to apply a given

evaluation rule and demonstrate progress. For example if we reach a situation where we have one value applied to another, then we know that the first value must have an arrow type for this to be well-formed. Then, by canonical forms, we infer that since the first value is a value and it has an arrow type that it must be a lambda. This then allows us to say that we can apply the rule that substitutes the second value for the variable into the body of the lambda.

As a result of constructor equivalence along with constructors that are applications of one constructor to another or projections from other constructors, there are an infinite number of ways to form "different" constructors which are populated by types. For example, one could apply the identity function an arbitrary number of times to the type int and the result would be equivalent to type int. Since there are integers, the type would be populated by values (any integer constant). As we wish to say that any value is canonical at its type we have an infinite number of cases to deal with. There are many ways to approach this problem. One method might be to give canonical forms for canonical types and a reduction for other types and then prove that every type reduces to a unique canonical type using confluence. One could then define the canonical forms of a type to be the same as those of the canonical type that it reduces to.

We did not, however, follow this approach. Because of an issue that arose in preservation and progress (see section 4.5), we ended up defining an algorithmic version of constructor equivalence. This presented us with another option that avoided the need to define a reduction and prove confluence. Our canonical forms lemma asserted that if a value `v` is well-formed at type `t`, then `v` is canonical at some type `t'` and, furthermore, `t'` is equivalent to `t`. Without the algorithmic version of constructor equivalence, it is not clear that this is a useful lemma to prove. With the normal version of contructor equivalence transitivity makes it extremely difficulty to show any sort of structural relation between `t` and `t'`. The algorithmic version makes it much clearer that there is a structural relationship between equivalent types if they are both canonical types. As we only ever care about canonical forms at canonical types, this effectively gives us that `v` is canonical at `t` itself for all the cases that interest us.

4.5 Preservation and Progress

The proofs of preservation and progress have presented more difficulty than the others. As a result of type equivalence it is not possible to do simple inversion. One might expect that a sort of inversion lemma could be proved that marched through the type equivalence, found the desired results, and then modified them to fit the equivalent types, but it is not clear that such a lemma is proveable. An alternative is to provide an algorithmic equivalence relation for which the inversion lemmas are proveable and then prove that the algorithmic relation is equivalent to the version used by the IL by showing that it is sound and complete [3]. The problem with this approach is that the completeness proof for the algorithm is usually done with logical relations which are not supported by Twelf. While a syntactic proof by Mark Lillibridge does exist, the proof is significantly less elegant than the logical relations proofs. For the moment it is necessary to make the extra assumption that the algorithmic constructor equivalence relation is complete. With this assumption, the progress lemma goes through without additional difficulty, but another problem arises in preservation.

The second issue in preservation arises in the case when modules are being evaluated. When a piece of the module is fully evaluated down to a value, it is substituted into the rest of the module. The problem with this is that it is also substituted into the rest of the signature for the module. This results in a change in the signature while preservation states that the signature never changes.

While this problem has not yet been resolved, we believe we have a solution for it - the signature resulting from the substitution should be a sub-signature of the one before the substitution as it just stipulates what the variable needs to be. Showing this would be fairly simple with singleton-kinds, but is somewhat more complicated in the translucent sums version of the definition. This is yet another reason why we regret the choice to follow Harper-Stone[1] closely rather than switching to singleton kinds. We believe that this is still doable and will resolve the issue.

5 Harper-Stone Bugs

One of the results of this work that demonstrates the improvement over prior methods is the number of bugs found in the Harper-Stone definition[1] and the ways in which they were found. The sorts of bugs that appear in Harper-Stone are inherent when language definition is done by hand. At present there are 34 bugs in it that are explicitly marked in the Twelf version. This is not a perfect count as some of the bugs refer to multiple issues and some refer to larger problems than others. Also, some of the bugs are really duplicates of the same bug in similar cases. Due to our departure from Harper-Stone in the dynamic symantics it is entirely possible that some bugs were missed as we simply did things differently.

A number of the bugs were found during the initial process of formalizing the definition. In trying to rewrite the rules in LF some typos immediately become apparent as some of the rules just don't make sense as written. One of the more interesting bugs found while formalizing the definition was the use of an undefined judgement. One of the sub-signature rules requires that *dec* ≡ *dec'*. While it is clear what this means, the judgement is not actually defined, as Twelf is more than happy to point out. Most of the bugs found in this way could easily be found by a careful proof reading by enough people, but there are plenty of subtler bugs that would likely go unnoticed through many proof readings.

The majority of the interesting bugs were discovered while proving regularity. This is probably in part because regularity was done first and in part because regularity covers all of the static semantics. During the course of proving a theorem one thinks about different things and thinks about them in different ways than when one is formulating the rules. While writing the rules for sub-signatures, for example, one thinks to ensure that the first term in the first signature is the same as the first term in the second signature or that it is a sub-signature of it and that the rest of the first signature is a sub-signature of the rest of the second signature. It is less likely to occur to someone that the label for the first term in the signature had better not occur again, however it turns out that this condition is necessary for regularity to hold. As we would like only well-formed signatures to be sub-signatures of one another, we need to ensure that there are no duplicate labels when showing that one signature is a sub-signature of another.

Another issue that arises when proving the same regularity condition appears in the case where the first term is a module. Harper-Stone[1] tells us that `sub/sdecs ST (sdecs/mod L S SD) (sdecs/mod L S' SD') <- sub/sig ST S S' <- ({m:mod} ok/var/mod m S -> sub/sdecs ST (SD m) (SD' m))` which means that `sdecs/mod L S SD` is a sub-sdecs of `sdecs/mod L S' SD'` provided `S` is a sub-signature of `S'` and `(SD m)` is a sub-sdecs of `(SD' m)` when we assume `m` is well-formed at `S`. (Note: a collection of sdecs is basically a signature) The problem that occurs is that when we apply the inductive hypothesis, we find that `({m:mod} ok/var/mod m S -> ok/sdecs ST (SD m))` meaning that `(SD m)` is well-formed provided `m` is well-formed at `S` and also `({m:mod} ok/var/mod m S -> ok/sdecs ST (SD' m))` meaning that `(SD' m)` is well-formed provided `m` is

well-formed at `S`. The first of these conditions is exactly what we need to prove that `(sdecs/mod L S SD)` is well-formed. The problem is that the second will allow us to prove that `(sdecs/mod L S SD')` is well-formed but not `(sdecs/mod L S' SD')` as we have no way of changing the assumption. In general there is no reason to believe `(sdecs/mod L S' SD')` is well-formed given that `(sdecs/mod L S SD')` since we don't know that `S'` is a sub-signature of `S` and in general it isn't. Thus, we must additionally assume that `SD'` is well-formed in the presence of `m` when `m` is well-formed at `S'`.

While proving regularity caught most of the bugs in the static semantics, it did not catch any in the dynamic semantics. The bugs here that were not caught while formalizing the evaluation rules had to wait until preservation and progress. Progress primarily turned up bugs where evaluation rules had been omitted. These included various cases such as trying to evaluate a case statement and having no way of pushing it on the stack and trying to pop a projection frame and actually project from a module, but not having a rule to pop the frame off. There was, however, one interesting bug that turned up in progress. While trying to demonstrate that evaluation can continue when the top frame was a case statement, it was necessary to show that the expression we have evaluated was an injection into a sum type. Generally one would simply appeal to canonical forms, however case takes an unknown sum type while an injection has a known sum type. Thus it is not possible for case to be applied directly to an injection. As it turns out, there were no canonical forms at the unknown sum type because an expression had accidentally been left out of Harper-Stone. The missing expression coerces an injection to drop the knowledge of what portion of the sum it is so that it just has the general unknown sum type. After adding this expression, typing rules, evaluation rules, and filling in all the new cases in the various theorems, the proof was able to proceed.

Another problem with the same case arose in preservation. The evaluation rule states that the relavent expression for the current case is pulled from the list and applied to the value inside the injection, however the typing rules for a case statement require that all the expressions for handling various cases take the injection of `V` rather than `V` itself. Thus the result of the evaluation could not possibly be well-typed at the correct type as it is not well-typed at all.

In addition to all of the bugs in Harper-Stone that were caught by Twelf and by doing these proofs, many of our own bugs were caught. Every time we found a bug, we looked up the corresponding rule in Harper-Stone. About half the time the bug was present in Harper-Stone and about half the time the rule had been miscopied.

6 Poplmark Challenge

In addition to proving type-safety for Standard ML in Twelf, we have completed the Poplmark Challenge. The Poplmark Challenge was put out by the University of Pennsylvania and the University of Cambridge to gauge whether tools like Twelf are mature enough for proving meta theorems about programming languages. The challange asked for proofs of transitivity and reflexivity of algorithmic sub-typing and of type-safety for a small language called F-sub. The completion of this challenge along with the definition of and proof of type-safety for Standard ML demonstrates that the time is indeed right for computer checked proofs of meta theorems.

While F-sub is a much smaller language than Standard ML, the proof of transitivity provided new challenges that did not arise with Standard ML. This proof had an issue with dependent types

and being able to reorder assumptions in the context much like the issue that arose in substitution (see section 4.3). This time, however, the `hack/arg` trick was not applicable as there could be multiple variables that were simultaneously the argument. In this case we were holding onto type variables and the assumptions that they were sub-types of another type. As the dependencies only held between the variable for one pair of assumptions and the sub-typing assumption for the other, we realized that by holding onto only the sub-typing assumption we could reorder the context in a suitable way. The resulting issue is that variables in the context are separated from their sub-typing assumptions. This would allow the appearance of arbitrary sub-typing assumptions in the context which is problematic as arrow might be a sub-type of records, etc. We resolved this issue by making a new judgement called `var` which stated that the associated type was a variable. By requiring that `var t` always be placed in the context with `t` and requiring that `var t` be proveable before making assumptions about `t`, we were able to show that assumptions are only made about variables and to complete the proof.

7 Conclusion

While the proof of type-safety for Standard ML is not quite complete, we believe that it will be done within the next week. At present the proof of progress is complete and preservation is only missing a few cases. As mentioned in section 4.5, we believe we know how to extend the proof to cover these cases and thus have a complete type-safety theorem for Standard ML. It is our intention to persue this to the end. After completing this, the entire proof will be computer checkable except for the completeness of algorithmic equivalence and the elaborator which exist on paper.

Although Twelf is sufficiently rich as to be able to prove meta theorems like type-safety for real languages such as Standard ML, there is still plenty of room for improvement. Many of the problems that were encountered in doing these proofs could be eliminated by adding additional features to Twelf. The difficulty with dependent types in proving substitution could be eliminated by adding a feature that allows one to grab the rest of the context after the dependent variable or otherwise deal with variables that occur in the middle of the context rather than at the end. Another nice feature would be an ability to write theorem cases to cover variables and pull the pieces needed for the proof from the context rather than needing to prove the theorem separately for every variable declared. These features and others would greatly enrich the power of Twelf and the ease with which it can be used.

Despite these short comings, it is clear that machine checked proofs are not a dream in the future but rather a reality in the present. While this proof of type-safety for Standard ML is not yet quite complete, it provides a standard to which the field should rise to meet. When it is complete, it will also provide a great tool for defining other languages as they can be elaborated to the same IL with little or no additions. This will greatly reduce the amount of labor required to design new languages with useful properties like type-safety.

8 Future Work

Once the proof of type-safety for Standard ML is complete, it will not be completely machine checkable. At present the elaborator and proof of correctness for the algorithmic constructor equivalence only exist on paper. We do not believe that it would be difficult to write the elaborator

in Twelf and we intend to so. The algorithmic constructor equivalence, however, is best done with logical relations. We have been lead to believe that Delphin will provide support for logical relation proofs, and thus we intend to redo everything in Delphin when it becomes available. We also intend to depart from Harper-Stone and modernize the definition at the same time by using singleton kinds instead of translucent sums.

References

[1] Robert Harper and Chris Stone. *An Interpretation of Standard ML in Type Theory.* Technical Report. June, 1997.
[2] Chris Stone and Robert Harper. *A Type-Theoretic Interpretation of Stand ML.* Milner Festschrifft. 2000.
[3] Chris Stone and Robert Harper. *Decidable Type Equivalence in a Language with Singleton Kinds.* POPL. January, 1999.

Appendix A: The IL

```
%%%%%  Syntactic Classes  %%%%%

kind            : type.  %name kind K.
con             : type.  %name con C.
exp             : type.  %name exp E.
loc             : type.  %name loc L.
tag             : type.  %name tag T.
sig             : type.  %name sig S.
mod             : type.  %name mod M.

krow            : type.  %name krow KR.
crow            : type.  %name crow CR.  %% rdecs
erow            : type.  %name erow ER.  %% rbnds
fbnds           : type.  %name fbnds FB.
sdecs           : type.  %name sdecs SD.
sbnds           : type.  %name sbnds SB.

stp             : type.  %name stp ST.   %% store type

%%%%%  Kinds  %%%%%

kind/t          : kind.
kind/crecord    : krow -> kind.
kind/arrow      : kind -> kind -> kind.

krow/nil        : krow.
krow/cons       : label -> kind -> krow -> krow.

%%%%%  Constructors  %%%%%

con/int         : con.
con/char        : con.
con/record      : crow -> con.
con/ref         : con -> con.
con/arrow       : con -> con -> con.
con/tagged      : con.
con/tag         : con -> con.
con/sum         : crow -> con.
con/ksum        : label -> crow -> con.
con/fst         : mod -> label -> con.
con/lam         : kind -> (con -> con) -> con.
con/mu          : con -> con.
con/app         : con -> con -> con.
con/crecord     : crow -> con.
con/proj        : label -> con -> con.

crow/nil        : crow.
crow/cons       : label -> con -> crow -> crow.

%%%%%  Expressions  %%%%%

exp/loc         : loc -> exp.
exp/tg          : tag -> exp.
exp/fix         : crow -> (exp -> fbnds) -> exp.
exp/app         : exp -> exp -> exp.
exp/record      : erow -> exp.
exp/proj        : exp -> label -> exp.
exp/handle      : exp -> exp -> exp.
exp/raise       : con -> exp -> exp.
```

```
exp/ref         : con -> exp -> exp.
exp/get         : exp -> exp.
exp/set         : exp -> exp -> exp.
exp/roll        : con -> exp -> exp.
exp/unroll      : exp -> exp.
exp/injs        : crow -> label -> exp -> exp.
exp/projs       : con -> label -> exp -> exp.
exp/case        : con -> exp -> erow -> exp.
exp/newtag      : con -> exp.
exp/tag         : exp -> exp -> exp.
exp/iftag       : exp -> exp -> exp -> exp -> exp.
exp/inteq       : exp -> exp -> exp.
exp/chareq      : exp -> exp -> exp.
exp/snd         : mod -> label -> exp.
exp/int         : int -> exp.
exp/char        : nat -> exp.

erow/nil        : erow.
erow/cons       : label -> exp -> erow -> erow.

fbnds/nil       : fbnds.
fbnds/cons      : con -> con -> (exp -> exp) -> fbnds -> fbnds.

%%%%%  Locations  %%%%%

loc/n           : nat -> loc.

%%%%%  Tags  %%%%%

tag/n           : nat -> tag.

%%%%%  Stores  %%%%%

stp/nil         : stp.
stp/loc         : con -> stp -> stp.
stp/tag         : con -> stp -> stp.

%%%%%  Signatures  %%%%%

sig/sig         : sdecs -> sig.
sig/arrow       : sig -> (mod -> sig) -> sig.

%%%%%  SDecs  %%%%%

sdecs/nil       : sdecs.
sdecs/con       : label -> kind -> (con -> sdecs) -> sdecs.
sdecs/coneq     : label -> kind -> con -> (con -> sdecs) -> sdecs.
sdecs/exp       : label -> con -> (exp -> sdecs) -> sdecs.
sdecs/mod       : label -> sig -> (mod -> sdecs) -> sdecs.

%%%%%  Modules  %%%%%

mod/struct      : sbnds -> mod.
mod/lam         : sig -> (mod -> mod) -> mod.
mod/app         : mod -> mod -> mod.
mod/proj        : mod -> label -> mod.
mod/ascribe     : mod -> sig -> mod.
```

```
%%%%%  SBinds  %%%%%

sbnds/nil       : sbnds.
sbnds/con       : label -> con -> (con -> sbnds) -> sbnds.
sbnds/exp       : label -> exp -> (exp -> sbnds) -> sbnds.
sbnds/mod       : label -> mod -> (mod -> sbnds) -> sbnds.

%%%%%  Derived Forms  %%%%%

%% We define the unit type to be the empty record.
con/unit        : con = con/record(crow/nil).
exp/unit        : exp = exp/record(erow/nil).
```

This is not the complete IL as it mentions neither stores nor stacks. The complete proof is available through CVS. Everything shown here is either a type or a constructor. The types are at the top and are denote as `foo:type`. The constructors indicate the type the produce on the right and the arguments that they take to create that type, separate by arrows to the left. Thus `con/lam :  kind -> (con -> con ) -> con.` means that `con/lam` takes a kind and function from constructors to constructors and produces a new constructor.

Appendix B: Regularity

```
il/reg/exp_con : ok/exp ST E C -> ok/con C kind/t -> type.
mode il/reg/exp_con +Eok -Cok.

-: il/reg/exp_con
    (ok/exp/var
       Cok
       _)
    Cok.

-: il/reg/exp_con
    ok/exp/int
    ok/con/int.

-: il/reg/exp_con
    (ok/exp/app
       (E'ok : ok/exp ST E' C')
       (Eok : ok/exp ST E (con/arrow C' C)))
    Cok
    <- il/reg/exp_con
        Eok
        (ok/con/arrow
             (Cok : ok/con ST C kind/t)
             (C'ok : ok/con ST C' kind/t)).
```

This is the theorem statement for one of the regularity conditions and a few of the cases from the proof. There theorem statement at the top says that it is a relation between proofs that `E` is well-formed at type `C` and that `C` is a well-formed type. The next line indicates that the proof that `E` is well-formed is an input and the other proof is an output. The cases below show various ways of generating the output from the input. The first case just looks up the input inside, the second case knows what the right output is and makes it up on the spot, and the third case makes an inductive call and analyses the results.