Reflections on the Design of a Specification Language

Stefan Kahrs* and Donald Sannella**

Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ

Abstract. We reflect on our experiences from work on the design and semantic underpinnings of Extended ML, a specification language which supports the specification and formal development of Standard ML programs. Our aim is to isolate problems and issues that are intrinsic to the general enterprise of designing a specification language for use with a given programming language. Consequently the lessons learned go far beyond our original aim of designing a specification language for ML.

1 Introduction

There are many different approaches to the problem of producing correct software systems in a given programming language. One line of attack involves the use of a specification language that is tailor-made to specifying and verifying properties of programs written in that particular programming language. This typically involves the use of a logical language that is appropriate for writing assertions about entities arising in programs written in that programming language. Some examples are: Anna [LvH+87] for use with Ada; Larch [GH93] adapted to the programming language in question via use of an appropriate "interface language", e.g. Larch/C++ [Lea96]; and our favourite, Extended ML [KST97] for use with Standard ML. Closely related is work on logics for reasoning about programs written in particular programming languages, e.g. Haskell [Tho93]. Although most of the details of this enterprise are specific to the particular programming language at hand, certain problems and issues are common to all programming languages or to a class of languages.

In this paper, we reflect on our experiences from work on the design and semantic underpinnings of Extended ML with emphasis on some of the more general lessons learned. The topics we cover range from the very general to the somewhat specific: Sect. 4 on the relationship between models of programs and models of specifications applies to any programming language; Sect. 5 on adding logical formulae to a language with a Hindley-Milner (implicitly polymorphic)

* Now at Computing Laboratory, University of Kent, Canterbury CT2 7NF. E-mail smk@ukc.ac.uk. This research was supported by EPSRC grant GR/K63795.

** E-mail dts@dcs.ed.ac.uk. This research was supported by EPSRC grant GR/K63795, an EPSRC Advanced Fellowship, an SOEID/RSE Support Research Fellowship and the EC-funded FIREworks working group.

type system is relevant to any programming language having such a type system; most of Sect. 6 on indistinguishability is relevant mainly to ML and fragments of ML. We begin with a brief description of Extended ML to provide some context for the rest of the paper.

2 Extended ML in brief

Extended ML (EML) is a wide-spectrum language for the specification and development of modular Standard ML (SML) programs. "Wide-spectrum" means that it encompasses both specifications and programs, as well as hybrids between the two. These hybrids arise as the intermediate stages of the process that turns a formal specification into a concrete program that implements it.

EML was conceived in the mid-1980s [ST85], combining ideas from algebraic specification and the then rapidly evolving functional programming language ML. Once ML was standardised and given a formal semantics in 1990 [MTH90], a project was set up to do the same with EML, resulting in its formal definition in 1994 [KST94].

We are not going to describe the features of EML in any but the most superficial detail. See [KST97] for more details and a gentle but thorough introduction to the EML semantics. A programmer-oriented introduction is [San91].

We can roughly describe EML as an extension of SML (minus some of its imperative features) with the following specification features:

- placeholders for expressions, type expressions, and *structure*[1] expressions; these are used to express incomplete programs, which are useful entities during program development
- axioms in structures; these are used to narrow down the possible choices for replacing placeholders
- axioms in *signatures*[1]; these demand and/or export properties of the implementing structure
- first-order logic with equality as the language for axioms.

This is a gross simplification and we shall have to expand on some of this later on. The definition of EML [KST94] is an extension of the definition of SML [MTH90] by (among other things) a definition of the meaning of axioms and what it means for a structure to satisfy the axioms in a signature.

3 Fundamental principles

Suppose we are given a programming language P and the task of designing a specification language S suitable for the specification and development of P-programs.

[1] "Structure" is ML-speak for module, "signature" for module interface.

Is this always possible? Which features should S contain, which primitives, which logical connectives? Equally importantly: which features should S *not* contain? To a certain extent one can answer these questions generically.

Different specification languages have different aims. Near one extreme would be a specification language that is intended as a formal notation for documenting programs, or as a vehicle for requirements capture, with no way to verify with any degree of formality that a given program satisfies a given specification. Then there is no need to make a formal connection between P and S, and indeed S may be appropriate for a range of programming languages. Near the opposite extreme would be specification languages like EML where a central aim is to enable proofs about specifications, and proofs that a given program satisfies a given specification. Here a formal connection between P and S is essential to establish the soundness of inference rules used in proofs that connect P-programs and S-specifications. Our concern in this paper is with specification languages of the latter kind.

Given that aim, it is not possible to come up with a meaningful specification language for P unless P has a formal semantics. Without a formal semantics for P we are not certain what P-programs are supposed to do, making it impossible to establish reliably any property of any P-program or to prove interesting relationships between P-programs and S-specifications. Unfortunately, this requirement rules out most present-day programming languages.

The design of S is constrained by the properties of the semantics of P. For example, the properties of P-programs we can express in S should not transcend the properties we can establish from the formal semantics of P. This is closely related to the reason why we need a formal semantics for P in the first place.

For instance, the dynamic semantics of SML [MTH90] defines the result of evaluating an expression in a particular environment and a given state. But it does not specify the required time and space resources for such an evaluation. The size of the derivation of the evaluation judgement (built from instances of the rules of the semantics) indicates the required resources *in a naive evaluation model*, but this information is unreliable — SML compilers are not forced to stick to the evaluation model implicitly suggested by the SML semantics and hardly any of them do so. This means that any specification language for SML should abstain from specifying the efficiency and/or complexity of a program.

One may object: people do reason about the efficiency of SML programs, don't they? But if compilers are allowed to modify the performance of a program by optimising it (which in some cases may even slow it down) then the observed performance becomes compiler-dependent. In other words: efficiency is a property of the machine program the compiler chooses to realize a source program, rather than a property of source programs themselves. When we reason about the efficiency of programs we assume that the compiler is not clever enough to significantly depart from the naive evaluation model given by the operational semantics. There is no formal justification for such an assumption.

If P is a typed language, it is natural to exploit its type system both to coordinate the required link with S and to provide the basis of a type system

for S. Although the utility of a type system for specifications as such is a matter of some debate — see e.g. [LP97] — we can hardly avoid mentioning types in S when asserting properties of typed programs in P. For example, when specifying the behaviour of a function $f : t \to t'$ it is often necessary to quantify over the values of type t. Apart from this, there is also the important design issue of making P-programmers feel "at home" when writing S-specifications. It therefore seems desirable that the type system for S be as close as possible to the type system for P. When, as in the case of EML, P is a subset of S, the type system of S should be a *conservative* extension of the type system of P: a P-expression e has a P-type t in S iff e has type t in P.

4 Models of programs vs. models of specifications

The semantics of the programming language P will assign *models* to programs of P. For each P-program p, its model $[\![p]\!]$ will contain some assortment of mathematical objects modelling the components of p, including (for example) the functions defined by p.

Any specification language S needs a semantics which defines the meaning $[\![s]\!]$ of each S-specification s. This is a necessary basis for specification-based proof: proof that a given program satisfies a given specification; proof that one specification is a *refinement* of another; or proof that all programs satisfying a given specification will satisfy a given property. When we design a specification language S for use with a programming language P, it is natural to define the meaning of an S-specification as the class of all P-models (i.e. models of well-formed P-programs) having the indicated components and satisfying the requirements spelled out in the specification (see e.g. [ST97]). This enables us to say that a P-program p satisfies an S-specification s exactly when the model of p is in the class of models determined by s: $[\![p]\!] \in [\![s]\!]$.

The expressiveness of P dictates the structure of models of P-programs. For instance, if P provides constructs for defining non-deterministic functions, models of P-programs containing such functions will need to model them using something more exotic than ordinary set-theoretic functions. Even if P does not provide such constructs, provided P is sufficiently expressive (that is: unless it is extremely inexpressive), functions in P-programs cannot be modelled by arbitrary set-theoretic functions. For example, the untyped λ-calculus requires a domain D of values such that $D \cong D \to D$; here $D \to D$ cannot be the whole function space (since $D \cong D \to D$ implies $|D| = |D \to D| = |D|^{|D|}$ i.e. $|D| = 1$) so it is taken to be the space of *continuous* functions [Gun92]. Another source of restrictions on models of P-programs is the desire to reflect more accurately the constraints that P imposes. For instance, no matter what P is, no P-program will contain definitions of non-computable functions and so it would be natural to take only *computable* functions in P-models. In SML, each function is modelled as a *closure* which contains the expression used in defining the function, so we get only the SML-expressible functions [MTH90]. Of course,

all of these are computable, but not all computable functions of a given type are SML-expressible [Kah96].

Putting these together (the decision to interpret S-specifications using classes of P-models and the imposition of computability and other restrictions on P-models) leads to a possible problem, as the following example from [ST96] illustrates.

Example 1. Let $\varphi_{\texttt{equiv}}$ be a sentence which asserts that `equiv(`n`,`m`) = true` iff the Turing machines with Gödel numbers n and m compute the same partial function (this is expressible in first-order logic with equality, since the equivalence of TMs is arithmetical [Rog67]). Now consider the following specification:

```
local val equiv : nat * nat -> bool
      axiom φequiv
in val opt : nat -> nat
   axiom forall n:nat => equiv(opt(n),n) = true
end
```

This specifies an optimizing function `opt` transforming TMs to equivalent TMs. (Axioms could be added to require that the output of `opt` is at least as efficient as its input.) If functions in P-models are required to be computable (and the semantics of specifications is compositional with models of *local* s *in* s' *end* obtained by forgetting the s-components of models of $s; s'$) then this specification will have *no* models because there is no computable function `equiv` satisfying $\varphi_{\texttt{equiv}}$. Yet there *are* computable functions `opt` having the required property, for instance the identity function on `nat`. Thus this specification disallows P-programs that provide exactly the required functionality. □

The example is expressed in terms of Gödel encodings of Turing machines where its practical utility may not be apparent, but exactly the same example could be phrased in terms of program fragments in a real programming language and a specification like the one above (and exhibiting exactly the same problem) could then appear as part of the specification of an optimizing compiler or program transformation system.

Here are three ways around this problem:

1. Treat local functions differently from "exported" functions, allowing them to be non-computable. Programs are not required to implement local functions in specifications anyway.
2. Relax the computability requirement on all functions.
3. Prohibit local functions in specifications.

The second solution seems simpler than the first because it is uniform. This is the approach taken by EML, where each function is modelled as an *EML-expressible* closure — still a closure, but where the expression in the closure is allowed to include "logical" constructs such as universal and existential quantifiers rather than being expressible using just the constructs of SML [KST94,KST97]. The third solution is unattractive since it sacrifices a great deal of expressive power.

Relaxing conditions on models needs to be done with care. Restrictions needed to ensure that models exist are still required (see the discussion of the untyped λ-calculus above). And there is a "logical" limit on expressibility: provided S extends Peano arithmetic, Gödel's fixpoint theorem can be applied to show that if satisfaction of the closed formulae of S can be defined in S itself (e.g. as a total function of type `formula -> bool`), then S is necessarily inconsistent.[2] It appears that any attempt to define EML satisfaction in EML yields a function that fails to terminate in some cases.

5 Parametricity

The kernel type system of most functional programming languages these days is Hindley-Milner polymorphism [Mil78], i.e. shallow, implicit polymorphism. ("Shallow", means that all type quantifiers occur outermost; "implicit" means that type abstraction and application are syntactically suppressed.) SML needs some modifications to cope soundly with imperative features, but we can ignore this for the moment.

The implicitness of type abstraction and type application strongly limits the options for possible extensions of the type system, should an extension be required to accommodate the specification logic: type inference and type checking for System F are undecidable [Wel94], as is type inference for Hindley-Milner polymorphism with the addition of proper polymorphic recursion [KTU93].

5.1 Prerequisites for implicit polymorphism

Why do we get away with implicit polymorphism? That is, why are we satisfied with the particular choices of type abstraction and type application selected by the type inference algorithm?

There are two fundamental reasons why this is so:

1. There is a best possible choice — and the type inference algorithm picks it.
2. Whatever choice is made, the outcome of evaluation is not affected.

The mentioned best possible choice is the so-called "principal" or "most general" type. The principal type subsumes all other possible types, in a technical sense which we can ignore here. In a certain sense, choosing the principal type is like[3] making no choice at all, leaving all options open.

The second reason is much more important.

Since type applications are implicit, the types inferred for expressions by the type inference algorithm are to a certain degree arbitrary. Consider the inference rule for type-checking function application:

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau}$$

[2] Thanks to Martin Hofmann for this observation.

[3] There are a couple of involved technical reasons why this is not quite true for SML, even after the 1997 revision [MTHM97]. For our purposes this is a side-issue.

When we infer the type of an application term $e_1\ e_2$, the rule requires that the argument type of the function e_1 and the type of the actual parameter e_2 agree — these are the two occurrences of σ in the premise of the rule. The type inference algorithm makes sure that this is the case, but this does not necessarily completely determine the type σ, since it is possible that different types have this property. We would not want these arbitrary choices to influence the computation in any way.

Other arbitrary choices arise when type variables are implicitly abstracted at declaration level. *All* type variables are abstracted that can possibly be abstracted (i.e. those in the type that do not occur free in the context), and the order of abstraction is arbitrary. Again, these arbitrary choices should not influence the computation.

There are several (related) ways of capturing this idea, e.g. Reynolds' notion of *parametricity* [Rey83] and Wadler's *theorems for free* [Wad89]. Essentially, type quantification can be manipulated in this implicit manner because types do not interfere with computation in System F.

More concretely, one can view type inference as a process that inserts *type conversion* functions, in addition to type abstractions and applications, whenever necessary to generate an explicitly typed program. Parametricity requires that these conversions are isomorphisms; this may not be the case — see [Cos92] — but for purposes of evaluation, verification and analysis of programs it is sufficient if they *behave* like isomorphisms: in other words, the function that converts back and forth should be *indistinguishable* from the identity function. (An informal understanding of indistinguishability will suffice for now. See Sect. 6.1 below for a definition.)

From what we have already seen, it should be clear that we need isomorphisms $\forall\alpha.\forall\beta.\tau \cong \forall\beta.\forall\alpha.\tau$ since we abstract type variables in an arbitrary order, and $\forall\alpha.\tau \cong \tau$ (if $\alpha \notin \mathrm{FV}(\tau)$) since we only abstract type variables that occur. There are more such requirements[4], but these two are sufficient to make our points.

It is not difficult to formulate the required isomorphisms in System F. We will write $\Lambda\alpha.t$ and $t[\tau]$ to denote type abstraction and type application on term level, respectively.

We can express the commutativity isomorphism (in both directions) by

$$\iota = \lambda x : (\forall\alpha.\forall\beta.\tau).\Lambda\gamma.\Lambda\delta.x[\delta][\gamma]$$

It is easy to check that $\iota \circ \iota$ is $\beta\eta$-convertible[5] to the identity function, and so ι is an isomorphism. More problematic is the conversion between τ and $\forall\alpha.\tau$ (with

[4] Another one is $\forall\alpha.(\tau_1 \times \tau_2) \cong (\forall\alpha.\tau_1) \times (\forall\alpha.\tau_2)$ which is needed since SML supports simultaneous declarations.

[5] To be precise, for call-by-value languages such as SML we need to restrict $\beta\eta$-conversion to *values*, as in Moggi's λ_c-calculus [SW96]. Under this restriction, we can reasonably assume that $\beta\eta$-convertible expressions are indistinguishable, even when the language is extended.

$\alpha \notin \mathrm{FV}(\tau)$). The required maps are

$$\iota_1 = \lambda x : \tau.\Lambda\alpha.x \qquad \iota_2 = \lambda x : (\forall\alpha.\tau).x[\mathbf{1}]$$

where $\mathbf{1}$ is the unit type (or any other chosen type). Again, $\iota_2 \circ \iota_1$ can easily be seen to be $\beta\eta$-convertible to the identity. However, while $\iota_1 \circ \iota_2$ is $\beta\eta$-convertible to $\lambda x : (\forall\alpha.\tau).\Lambda\beta.x[\mathbf{1}]$, it is not convertible to the identity function. We therefore require that $\lambda x : (\forall\alpha.\tau).\Lambda\beta.x[\mathbf{1}]$ is indistinguishable from the identity function whenever α is not free in τ. In this case, the required property can be proven in an extension of System F with "Axiom C" from [LMS93]. Turning this back into English: if the type of a term t does not depend on the type parameter then neither should the value of t itself be affected by it.

Extensions of the purely functional sublanguage with other features should preserve the property that $\iota_1 \circ \iota_2$ is indistinguishable from the identity function. This requirement applies to various forms of language extension including an extension with logical formulae or imperative features.

5.2 Assessing the logic

Quantification over values in a typed language P is itself necessarily typed, i.e. we quantify over values of a particular type. For example, if we specify the `reverse` function for lists then we are not concerned with what it would do if applied to numbers or functions — the type system of P is supposed to prevent that.

There is a problem with typed quantification which arises from the fact that the truth value of a formula may depend on the type of quantification. The simplest example is the following:

```
forall x:t => false
```

This formula is `false`, *unless* `t` is an empty type, in which case it is `true`. In EML, we view a type as empty if it has no values. There are indeed empty types in SML and EML, e.g. `datatype t = C of t`.

The example is perhaps unconvincing, first because in languages with lazy evaluation one would normally regard $\perp$ as inhabiting any type, and second because it resembles the empty-sort problem in algebraic specification [GM85,PW84] which can be dealt with by banning empty sorts altogether, considering that their usefulness for specification and programming is rather limited. However, the problem goes deeper than that; consider the following EML formula:

```
forall (x,y:t) => x == y
```

(Here, `==` is EML logical equality, see Sect. 6.) Again, this formula is `true` if `t` is an empty type, but it is also `true` if `t` is a singleton type, like `unit`. In general, first order logic with equality allows one to distinguish finite types from infinite types and also finite types of different cardinality.

The above example shows that the problem already appears for universally quantified equations. Here is another example of the same thing, which relies on the use of a function:

forall `(xs: t list) => rev xs == xs`

If `rev` is ordinary list reversal then this formula implicitly specifies the same property as the previous one: `x==hd[x,y]==hd(rev[y,x])==hd[y,x]==y`.

These examples show that the truth value of a logical formula can depend on the type of quantification. Indirectly, this means that its truth value can depend on the assignment of types to type variables, and therefore formulae for which this assignment can vary may have varying truth values. Since the type of a formula is just `bool`, the addition of typed quantification breaks the required isomorphism between `bool` and $\forall\alpha.$`bool`.

After making this observation it should not come as a surprise to observe that implicit polymorphism has some rather uneasy interactions with formulae. These do not occur often; in EML one has to employ the available forms of explicit polymorphism to contrive unpleasant examples. Here is one:

type `'a dummy = bool`
val `b:'a dummy =` *forall* `(x,y:'a) => x==y`

The variable `b` is bound to a boolean value, but is it `true` or `false`? Morally, this should depend on the type application at each instance of `b`, being `true` iff the argument type has at most one element. But type application is implicit in ML and in this case we cannot reconstruct what the type argument is as it is not retained by our implicit conversion ι_2.

The problem is aggravated by the identification of formulae and boolean expressions. As a consequence of this identification, formulae can appear within arbitrary expressions, exporting the observed type dependency to values of all types.

Of course, one can argue that in view of the evident type-dependency of the logic one should abandon implicit polymorphism and make all type abstractions and applications explicit. There are just two problems with this: firstly, implicit polymorphism is such a successful design feature because it combines the benefits of a strong type system (soundness) with the benefits of an untyped language (you do not have to write types); secondly, an explicitly typed wide-spectrum language would co-exist rather uneasily with an associated implicitly typed programming language.

EML sidesteps the problem of type-dependency by giving type-dependent expressions in axioms *no* value, and taking an *arbitrary* choice from among the possible values of type-dependent expressions that are not within axioms. The solution for axioms is satisfactory because we are concerned only with whether axioms are *satisfied* or not, and when an axiom has no value it is regarded as not being satisfied. The solution for type-dependent expressions outside axioms is less satisfactory but it seems adequate for practical purposes since this situation is very rarely encountered.

5.3 Imperative features

At this point it is perhaps worth pointing out that the addition of logical features is not the only language extension that sits uneasily with implicit polymorphism.

It is well known that imperative features such as references endanger the soundness of a polymorphic type system [Dam85,Tof88,Wri95]. What is perhaps less well-known is that the associated problems can largely be attributed to the *implicitness* of the polymorphism. One of the proposals in Xavier Leroy's thesis [Ler92] to circumvent the known soundness problems with polymorphic references is to make type abstraction explicit. Technically, this is achieved by having two different kinds of `let`-binding, a polymorphic one and a monomorphic one; whenever a `let`-bound variable is used which originates from a polymorphic `let` then we have an implicit type application which — in Leroy's suggested semantics — forces a new evaluation of the associated expression. In other words, Leroy only makes the type abstraction explicit; for his purposes he does not need to know the type parameter of the type application, the fact that there is *some* type application is sufficient.

The key to Leroy's idea is that type abstractions are treated as value abstractions — the evaluation of the body of the abstraction is delayed until a parameter is provided. The problem that this circumvents is again the fact that $\iota_1 \circ \iota_2$ is not indistinguishable from the identity function in System F extended with references. But this time it *is* $\beta\eta$-convertible to the identity function if η-conversion is unrestricted, which shows that unrestricted η-conversion is unsound in the presence of side-effects. There is a difference between passing t and $\Lambda\alpha.t[\alpha]$ as a parameter: for t we generate references once, for $\Lambda\alpha.t[\alpha]$ we generate references at each type application separately.

6 Logical equality and indistinguishability

The concept of equality is much underestimated in mainstream mathematics. The general attitude seems to be: what could be simpler than that? Consequently, equality is viewed as a primitive concept in set theory, in universal algebra, and thus in algebraic specification. In type theory, equality is no longer a primitive concept, and some type theories even have more than one notion of equality.

Considering the origins of EML in algebraic specification, it should not come as a surprise that the EML logic contains $t{=}{=}u$ (so-called *logical equality*) as one form of atomic formulae. Logical equality is defined using the notion of *indistinguishability* in order to make it extensional on function types [KST97]. Some problems that arise with the use of indistinguishability are discussed in the rest of this section.

6.1 Indistinguishability

Two expressions exp_1 and exp_2 of type τ are indistinguishable if and only if for any context $C[\,]$ of type `unit` with a hole of type τ we have that the evaluation of $C[exp_1]$ terminates iff the evaluation of $C[exp_2]$ terminates (cf. e.g. [Ong95]). One can always distinguish "genuinely" different values (such as 0 and 1, or `true` and `false`) in this way.

The choice of `unit` as the result type for the distinguishing contexts is somewhat arbitrary, except that expressions of type `unit` are *only* distinguished by their termination behaviour. An obvious alternative would be `bool` and distinguishing contexts returning `true` and `false`, respectively. If $C[exp_1] = \texttt{true}$ and $C[exp_2] = \texttt{false}$ then there is obviously a context C' that distinguishes exp_1 and exp_2 in the above sense. However, with this choice we would not even be able to distinguish the totally undefined function `undef` from any other function. This would make indistinguishability non-transitive since we would have f==`undef`==g for any two functions f, g having the same type. But in a system like the typed λ-calculus where evaluation always terminates, this choice is perfectly reasonable.

Indistinguishability is a difficult relationship. Clearly, it is not decidable, but worse than that, it is neither semi-decidable nor co-semi-decidable. The theory of β-conversion (for untyped λ-calculus) is not recursive either, but at least it is r.e. and so there is a complete proof system that enables us to establish β-convertibility whenever it holds. With typed λ-calculi (without general recursion) we typically have that indistinguishability is co-r.e., because we can enumerate the distinguishing contexts and evaluation always terminates,[6] but not r.e., because equality in the fully abstract model is undecidable; thus, we can at least have a proof system to *refute* indistinguishability in that setting.

For a system like SML where termination is not guaranteed, we have the worst of both worlds: indistinguishability is defined in terms of distinguishing contexts (which we can enumerate), but even if the context $C[\,]$ is given, comparing the termination behaviour of $C[exp_1]$ and $C[exp_2]$ requires a solution to the halting problem, in general.

Proposition 1. Indistinguishability for SML is neither semi-decidable nor co-semi-decidable.

Proof. Suppose that indistinguishability were co-semi-decidable, i.e. that distinguishability were semi-decidable. Then we would be able to decide the halting problem for any program t by comparing the constant c with the expression $(t; c)$ (which computes t, throws it away and then returns c). This solves the halting problem for t, because we can distinguish c from $(t; c)$ iff t fails to halt: if t fails to halt then the semi-decision procedure would terminate; so we just have to run the evaluation of t in parallel and wait to see which process terminates first.

Suppose that indistinguishability were semi-decidable. A similar argument applies, where we compare t with a looping program. □

The above argument can easily be adapted to languages other than SML.

Proposition 1 says that whatever proof system we may come up with to support formal proofs and/or refutations of indistinguishability, it will be incomplete in the strong sense that there will be indistinguishable expressions which the system will fail to certify as indistinguishable as well as distinguishable expressions which it will fail to distinguish.

[6] Since evaluation always terminates, here we define indistinguishability via contexts of type `bool` returning `true` or `false`.

6.2 Indistinguishability in the presence of impure features

The computational sublanguage of EML is not quite a "pure" functional programming language. Although references and input/output were omitted, exceptions were retained in the hope that they would not upset reasoning about programs too much. This hope turned out to be misplaced.

One aspect of ML exceptions can be expressed by so-called *names*. Names can be generated using a `new` function, and names can be compared for equality where separately-generated names are not equal. The type of names can be implemented in SML in various ways: using references (e.g. via `unit ref` and the equality of references, or an `abstype` with a local counter); or just via exceptions:

```
abstype name = A of exn * (exn -> bool)
with fun new f = let exception X
                 in f(A(X,fn X=>true| z=> false))
                 end
     fun eq(A(_,f),A(e,_)) = f e
end
```

As type of `new` we have `(name->'a)->'a` which exactly corresponds to the idea that `new` is a variable binder, i.e. we write `new (fn x=>a)` for the expression a with the new name `x`.

Reasoning about indistinguishability of programs containing names is notoriously difficult [PS93], to the point where for certain "obviously" equivalent expressions, no syntactic method of establishing indistinguishability is known. So, this is already bad news. What makes the situation worse is that the very presence of names affects indistinguishability of ordinary applicative programs:

Example 2. Consider the functions `g1` and `g2`, defined as follows:

```
fun g1 f x = (f x, f x)
fun g2 f x = let val z=f x in (z,z) end
```

The functions `g1` and `g2` are clearly indistinguishable in a purely applicative call-by-value language.[7] However, in the presence of names we can write the function `C` which distinguishes them: `C g1` is `false` while `C g2` is `true`.

```
fun C g = eq(g new (fn x=>x))
```

□

Notice that the result of post-composing `g1` or `g2` with either the first or second projection is indistinguishable (even in the presence of names) from ordinary function application. If follows that pairing is not a categorical product.

In EML we do not have names as a primitive; we have exceptions which — as we have seen — are expressive enough to encode names. However, they are

[7] In our examples, we nonchalantly claim the indistinguishability of particular expressions in certain sublanguages of SML without proof. One could establish these formally by using and combining techniques from the literature, e.g. applicative bisimulations [Abr90] and applicative equivalences [PS93].

more expressive than names, in the sense that the presence of exceptions allows even more applicative programs to be distinguished than names do.

Example 3. Consider the functions `andl` and `andr`, defined as follows:

```
fun andl a b () = if a() then b() else false
fun andr a b () = if b() then a() else false
```

The functions `andl` and `andr` are indistinguishable in applicative contexts and remain indistinguishable if we have names at our disposal. However, in the presence of exceptions we can write the function `C` which distinguishes them: `C andr` is `false` and `C andl` is `true`.

```
exception A
fun nothing() = raise A
fun ff() = false

fun C a = a nothing ff () handle _ => true
```

□

If we further add references and/or input/output then even more applicative programs can be distinguished. Here is an example of two functions that are indistinguishable in the presence of exceptions but distinguishable by references.

Example 4. Consider the functions `r1` and `r2`, defined as follows:

```
fun r1 g x = (g x; g x)
fun r2 g x = g x
```

The functions `r1` and `r2` are indistinguishable in the absence of references: if `g x` fails to terminate or raises an exception e then so will both `r1 g x` and `r2 g x`; otherwise the resulting values could only differ in freshly generated exception names, but then these sets of names are isomorphic. But the following function `C` distinguishes them: `C r1` is 3 and `C r2` is 2.

```
fun C r =
    let val x = ref 1
        fun g y = y := !y + 1
    in r g x; !x
    end
```

□

Fans of pure functional languages might be feeling smug at this point, since all our problems appear to be caused by just the features that are absent in pure languages. But this reaction misses an important point, since it is well-known that *monads* can be used to model imperative features within pure functional languages [Mog89], [PW93]. It follows that all the problems of reasoning about these features are already present in the applicative world.

Example 5. Consider the following definition of a "clock" monad:

```
type 'a clock = int -> 'a
fun unit x = fn _ => x
infix >>=
fun x >>= f = fn n => f (x n)(n+1)
```

This is not a monad in the categorical sense since the coherence laws do not hold, but for our purposes this does not matter. It is simply a program fragment which we could write if we were inclined to do so. The idea is to view the integer parameter of `clock` as the current time, so that each expression in the monad world can access the current time, while function application (>>=) makes time advance, because the result of application is interpreted one clock tick after the interpretation of the argument. One can use `clock` to simulate names as follows:

```
abstype name = A of int
with fun new f = f o A
     fun eq(A n,A m) = unit (m=n)
end
```

Now `new` and `eq` do not return `name` and `bool` but `name clock` and `bool clock`, respectively. We can now translate Example 2 into this setting:

```
fun g1' f x = f x >>= (fn r => f x >>= (fn s => unit (r,s)))
fun g2' f x = f x >>= (fn z => unit (z,z))
fun C' g = (g new (fn x=>x) >>= eq) 0
```

The functions `g1'` and `g2'` are the monadic translations of `g1` and `g2`, respectively. If the corresponding monad were the identity monad — with `unit` the identity function and >>= being reverse function application — then they would be identical to the original versions `g1` and `g2`. But in the clock monad, the function `C'` distinguishes `g1'` from `g2'`. □

The lesson we can learn from this is the following: if indistinguishability is such a difficult relationship in the presence of names that we have to resort to denotational methods to prove it, then it is just as difficult in the absence of names — we still need denotational methods. The only difference is that names pull the problem a few grades down the type hierarchy, making it omnipresent. But if we were looking for a *general* method to prove indistinguishability, one that operates on all types, then sticking to a pure language does not avoid the inherent problem. Indistinguishability is difficult!

Moreover it is a rather volatile relation: any additional features[8] chosen by an SML implementor for the library of his or her implementation may affect indistinguishability. As one of our earlier examples has shown, the indistinguishability relations of SML and EML differ because SML has references which can be used to distinguish otherwise equivalent applicative programs while these are absent in EML. Thus, SML programs developed in the EML formalism may only be partially correct in the sense that equivalences required in the specification which hold in the absence of references may fail to hold in their presence.

[8] A particularly nasty example would be an access to "system time", as this immediately makes any program optimisation invalid.

7 Conclusion

The design of EML unmasked more problems and issues than we have been able to cover above. We have concentrated on those that are of more general interest, and that are relatively easy to explain. One class of interesting issues that are rather difficult to explain in the space available involve SML's *module* language, concerning e.g. the interpretation of module interfaces and the treatment of module components that are not exposed by the interface. All of the issues discussed above pertain to SML's *core* language for defining the components of modules (types, values, etc.).

Some of the problems discussed above arose from our attempt to combine specification features with programming features *in a single language*. It is unclear to us whether all of the problems mentioned will arise if the specification and programming languages are decoupled as they are in the Larch "two-tiered" approach [GH93]. Our feeling is that the same problems, or some of them, may well re-emerge in a different form, but we have no concrete evidence for this assertion. Direct comparisons are difficult because the preliminary work on Larch/ML in [WRZ93] was (to the best of our knowledge) never followed up.

Prior to 1990, work on EML (by the second author and Andrzej Tarlecki) focussed on the use of ML-style modules in specification and formal development. The features of SML's core language and the specification constructs required at that level were viewed as one possible instantiation of this general picture [ST86]. Only when we looked at these features in excruciating detail while working on the semantics of EML did we discover problems like those described above. This makes us skeptical of attempts to connect specifications and programs on an informal level without reference to formal definitions of both languages. Even if the aim is not formal proofs of correctness, programming languages are complicated enough that there are bound to be hidden problems. Undertaking the detailed analysis that is required when writing a semantics appears to be the best way of exposing these.

Acknowledgements:

Our special thanks to Andrzej Tarlecki for long and productive collaboration on all aspects of EML and related topics, including some of the specific issues discussed above. Thanks to Michel Bidoit for useful comments on a draft.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[Cos92] R. di Cosmo. Type isomorphisms in a type-assignment framework. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 200–210, 1992.

[Dam85] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[GM85] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 38:173–198, 1985.

[Gun92] C. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[Kah96] S. Kahrs. Limits of ML-definability. In *Proceedings of PLILP'96*, volume 1140 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 1996.

[KST94] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, University of Edinburgh, 1994.

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[KTU93] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[LP97] L. Lamport and L. Paulson. Should your specification language be typed? Technical Report 425, University of Cambridge, Computer Lab, 1997.

[Lea96] G. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavorial Specifications*, pages 121–142. Kluwer Academic, 1996.

[Ler92] X. Leroy. Polymorphic typing of an algorithmic language. Rapports de Recherche No. 1778, INRIA, 1992.

[LMS93] G. Longo, K. Milsted, and S. Soloviev. The genericity theorem and parametricity in the polymorphic λ-calculus. *Theoretical Computer Science*, 121:323–349, 1993.

[LvH+87] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *Anna, a Language for Annotating Ada Programs: Reference Manual*, volume 260 of *Lecture Notes in Computer Science*. Springer, 1987.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proc. 4th IEEE Symp. on Logic in Computer Science*, pages 14–23, 1989.

[Ong95] C.-H. L. Ong. Correspondence between operational and denotational semantics: The full abstraction problem for PCF. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4*, pages 269–356. Oxford Univ. Press, 1995.

[PW84] P. Padawitz and M. Wirsing. Completeness of many-sorted equational logic revisited. *EATCS Bulletin*, 24:88–94, 1984.

[PW93] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th Symp. on Principles of Programming Languages*, pages 71–84, 1993.

[PS93] A. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. 18th Intl. Symp. on Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 1993.

[Rey83] J. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523, 1983.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[SW96] A. Sabry and P. Wadler. A reflection on call-by-value. In *Proc. Intl. Conf. on Functional Programming*, 1996.

[San91] D. Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Workshops in Computing, pages 99–130. Springer, 1991.

[ST85] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 67–77, 1985.

[ST86] D. Sannella and A. Tarlecki. Extended ML: An institution-independent framework for formal program development. In *Proc. Workshop on Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 364–389. Springer, 1986.

[ST96] D. Sannella and A. Tarlecki. Mind the gap! Abstract versus concrete models of specifications. In *Proc. 21st Intl. Symp. on Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1996.

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[Tho93] S. Thompson. Formulating Haskell. In *Proc. Workshop on Functional Programming*, Workshops in Computing. Springer, 1993.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1988.

[Wad89] P. Wadler. Theorems for free! In *Proc. 4th ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

[Wel94] J. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. 9th IEEE Symp. on Logic in Computer Science*, pages 176–185, 1994.

[WRZ93] J. Wing, E. Rollins, and A. Zaremski. Thoughts on a Larch/ML and a new application for LP. In *Proc. 1st Intl. Workshop on Larch*, Workshops in Computing, pages 297–312. Springer, 1993.

[Wri95] A. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–365, 1995.