A Semantics of Compile-time Reflection

James Hook and Tim Sheard*
Oregon Graduate Institute, Box 91000, Portland, Oregon 97291
{hook,sheard}@cse.ogi.edu

Abstract

A new language incorporating both ML-style type checking and a limited form of reflection is defined by giving an interpreter and showing how this interpreter may be interpreted as a compositional denotational semantics. The resulting language has a partial function as a compiler, but if the compiler terminates without a type error there will be no type-errors at runtime. Typing issues and first-class environments are discussed as well.

1 Introduction

Traditionally reflection and strong typing are found on different sides of the schism that divides the lisp and functional programming communities. This paper introduces Compile-time Reflective ML (CRML), a reflective language with compile-time type checking and run-time type safety[14]. CRML provides a bridge between two rich cultures.

When developing CRML, the goal was to create a system that was both expressive and useful, supported higher abstraction mechanisms, and that could be completely type checked at compile-time. The implementation largely succeeded in meeting these goals; this paper asks the question "What nature of beast have we created?"

As the name suggests, CRML supports only that reflection which can be resolved at compile-time. This paper presents a formal semantics of compile-time reflection. Its development has led to a better understanding of the CRML system, led to an improved understanding of reflective systems in general (and the reflective tower in particular), and led to generalizations of the CRML language that make it more uniform, and that allow it to solve problems not originally considered. These generalizations will be the basis for a second, improved implementation.

2 Why Reflection?

Reflection allows the implementation of abstraction mechanisms not possible in ordinary languages[11, 4, 5]. The most familiar use of reflection is the powerful notion of macro found in lisp dialects. As a simple example consider writing a single function that maps an n-ary function over n lists. In a typed language such a function is impossible because it cannot be typed. Yet given n, it is easy to imagine a function that computes a data structure representing the

*Hook was supported in part by a grant from the NSF (CCR-9101721). Both authors were supported in part by a contract with Air Force Material Command (F19628-93-C-0069).

```
datatype pat =                    datatype exp =
   Pvar of string                    Var of string
|  Ptuple of pat list             |  App of exp * exp
|  Pcon of string * pat           |  StringC of string
|  Pstring of string;             |  Abs of (pat * exp) list
                                  |  Tuple of exp list
                                  |  Quote of exp
                                  |  Escape of exp;
```

Figure 1: Mini-CRML Abstract Syntax

n-way map function. If we could reflect over this data structure obtaining the actual function we would have solved the problem.

Exploring reflection in typed languages has led us to a new paradigm of programming based on writing meta-programs that operate on types and the declaration of type constructors as well as writing macros that operate on expressions. For example, consider defining structural equality functions over freely constructed data types. It is easy to give "blackboard rules" for constructing the equality function systematically from the data type declaration. Reflection allows the rules to be written as a meta-program from the type of type declaration representations to the type of function declaration representations.

The description of the new paradigm of programming that CRML supports is beyond the scope of this paper. Applications of CRML are listed in the bibliography[13, 14, 12, 6].

3 A brief introduction to Mini-CRML

CRML is a variant of Standard ML. The implementation is based on Appel and MacQueen's SMLNJ[1]. This paper defines a small subset that illustrates the fundamental features of the language. Naively, CRML is a meta-language for Standard ML. Programs written in CRML are meta-programs. Meta-programs are programs that manipulate object programs. Object programs are written in an applicative subset of the "core" language of Standard ML.

In the Mini-CRML meta-language, the object-language is represented by two datatypes: `exp` for expressions, and `pat` for patterns. These types are given in Figure 1. Full CRML also represents types and declarations.

Programs manipulating program representations may either be expressed directly with the constructors or with an "object-language" extension to ML's syntax. Text within the object-language quotation brackets (`<< >>`) is parsed but not compiled. Its representation is returned as the value. Meta-language expressions may be included in the object-language text by "escaping" them with a backquote (anti-quotation) character (`). Examples illustrating the object-language notation are given in Figure 2. To disambiguate the types, the quotation bracket `<p< >>` is used for patterns.

Concrete syntax	Constructor based	Quote based
x	Var "x"	<< x >>
f x	App(Var "f",Var "x")	<< f x >>
	App(g,y)	<< `g `y >>
(x,y)	Tuple [Var "x", Var "y"]	<< (x,y) >>
	Tuple [x, y]	<< (`x,`y) >>
()	Tuple []	<< () >>
fn x => y	Abs(Pvar "x",Var "y")	<< fn x => y >>
	Abs(s,t)	<< fn `s => `t >>

Figure 2: Constructing Type and Expression Representations in CRML

To illustrate meta-programming, consider writing a single function that maps an n-ary function over n lists. When $n = 3$ we expect:

```
fun map3 f (Cons(x1,xs1),Cons(x2,xs2),Cons(x3,xs3))
             => Cons(f (x1,x2,x3), map3 f (xs1,xs2,xs3))
  | map3 f _ => [];
```

At the meta-level it is easy to describe the generation of this function for arbitrary n. To manage variable names in the development, we assume the two functions: add_index and iota. The add_index function concatenates its numeric argument to its string argument, and iota generates the list of integers between its two numeric arguments.

The meta-program generating the n-ary map follows. It builds the pattern for the first clause of the definition using the object language template <p<Cons(_ , _)>>, meta-language escapes to calculate the appropriate identifiers for the template, and the constructors Ptuple and Pvar to glue things together. The body of the definition is constructed similarly. Finally, an object-language expression serves as the template for the entire function definition.

```
fun mapN n =
let val nconses = Ptuple(map (fn n => <p<Cons(`(Pvar (add_index "x" n)),
                                              `(Pvar (add_index "xs" n)))>>)
                              (iota 1 n))
    val nheads = Tuple(map (fn n => Var(add_index "x" n)) (iota 1 n))
    val ntails = Tuple(map (fn n => Var(add_index "xs" n)) (iota 1 n))
in <<let fun mapNfun f `nconses => Cons(f `nheads, mapNfun f `ntails)
           | mapNfun f _ => []
      in mapNfun end>>
end;
```

Using this meta-program the simple object-program below calculates an equivalent declaration to the one given at the beginning of this example:

```
datatype value =
   stringV of string
|  closureV of (value -> value maybe) list
|  tupleV of value list
|  constructionV of string * value
|  constructorV of string;
```

Figure 3: Semantic Domain for Mini-CRML

```
val map3 = `(mapN 3);
```

In this example there is a clear distinction between the meta-language and the object language. In reality, both languages are the same: they are both CRML. In particular, this means that object-language brackets may appear in object-language expressions and meta-language escapes may appear in meta-language expressions. One consequence of this is that there is a potentially unbounded hierarchy of "meta-languages". The sections that follow describe how this power is restricted to a form that is meaningful at compile-time.

4 A semantics for Mini-CRML

This section develops a denotational semantics for CRML by first presenting a recursive interpreter that fails to be compositional and then showing how this interpreter may be stratified into a compositional denotational semantics.

The domain of values is described by the ML datatype declaration `value` in Figure 3. To simplify the examples, strings are used for identifiers, constructor tags, and basic constant values. It is important that there be a primitive type that can be interpreted as an identifier name. All other uses of strings could be satisfied by any countable flat domain.

Pattern based function definition (as found in HOPE and ML) is supported by representing closures as a list of functions from `value` to `value maybe`. Each element of the list corresponds to a separate clause of the function definition. Details of the encoding are given below. The semantics does not assign a meaning to functions that are not exhaustive; in this case the interpreter raises an exception in ML at runtime.

In the formal semantics it is assumed that a suitable domain has been constructed satisfying the domain equation implicit in the `value` datatype declaration. The details of the construction of this domain are not significant to the thesis of this work.

4.1 Expansion and Reification

The meaning of programs is determined in two separable phases, corresponding to elaboration time and run time in Standard ML. The first phase is called expansion and reification. It takes a mini-CRML program expressed with `Quote` and `Escape` and produces a quote free program in which all uses of the object-language quotation mechanism have been replaced by explicit constructions of values of type `exp` and `pat`. The second phase, evaluation, is a traditional ML interpreter augmented with a clause to define `Escape`.

Expansion and reification is defined by four functions, an expansion function that transforms expressions to expressions, and three reification functions for the types `list`, `pat`, and `exp`. The expansion function acts as the identity on all program fragments except those that are quoted, which it reifies. The reification functions each take an object of the appropriate type and return an expression that, when evaluated, will denote that object. For example, the reification of the pattern (x, y), which is represented in the abstract syntax as `PTuple([Pvar "x", Pvar "y"])` would be the expression:

```
App(Var "Ptuple", App(Var "Cons", Tuple [Pvar "x",App(Var "Cons",
                              Tuple [Pvar "y",App (Var "Nil", Tuple [])])]))
```

The definitions of the expansion and reification functions are given in Figure 4. Note that the only clause in the definitions of reify and expand that fails to be compositional (i.e. where the meaning of a phrase isn't expressed in terms of the meaning of its constituent subphrases) is in the clause defining the reification of a quoted expression. This is the first case in which the interpreter fails to be a valid denotational semantics.

4.2 Evaluation

The evaluation function, `eval`, takes two environments and an expression and produces a value. The first environment, called the global environment, is passed unmodified to all subexpressions; it makes the top level environment available to the reflection operators. This requires that the global environment for interpretation of functions include the types `exp` and `pat` and their associated constructor functions. The second environment is the traditional environment used to interpret variables. It is updated by the binding mechanism used in patterns and function definitions. The `eval` function and its elementary support functions are given in Figure 5.

As in Standard ML, functions are defined by a sequence of pattern-action pairs. The patterns are built out of constructors, free variables, and constants. If a value conforms to the pattern, the free variables in the pattern are bound to the associated components of the value. These bindings are available in the associated action, which is an expression. To compute the value of a function on an argument, the patterns are matched against the argument in the order in which they appear in the text. The value of the function is the value of the action corresponding to the first pattern that it matches.

To support this definition, the semantics must first assign a meaning to patterns. This is done with the function `bind`, which maps a pattern-value pair and an environment to a new environment, if the pattern matches, and which signals failure otherwise. This failure mechanism is provided by a maybe type.[1]

The meaning of patterns is used to construct the meaning of pattern-action clauses in the function `matchcl`. It takes a pattern-action pair and buils a function that uses `bind` to

[1] Following Spivey[15], `datatype 'a maybe = Just of 'a | Nothing`. SMLNJ calls this type `option`.

```
fun reifyl f [] = App(Var "Nil",Tuple [])
  | reifyl f (x::xs) = App(Var "Cons",Tuple[f x,reifyl f xs]);

fun reifyp (Pvar s) = App(Var "Pvar",StringC s)
  | reifyp (Ptuple l) = App(Var "Ptuple",reifyl reifyp l)
  | reifyp (Pcon(c,p)) = App(Var "Pcon",Tuple[StringC c,reifyp p])
  | reifyp (Pstring s) = App(Var "Pstring",StringC s);

fun reify (Var s) = App(Var "Var",StringC s)
  | reify (App(x,y)) = App(Var "App", Tuple[reify x, reify y])
  | reify (StringC s) = App(Var "StringC",StringC s)
  | reify (Abs(m)) = App(Var "Abs",reifyl (fn (p,e) => Tuple[reifyp p,reify e]) m)
  | reify (Tuple l) = App(Var "Tuple",reifyl reify l)
  | reify (Quote e) = reify (reify e)
  | reify (Escape x) = expand x

and expand (Var s) = Var(s)
  | expand (App(x,y)) = App(expand x, expand y)
  | expand (StringC s) = StringC(s)
  | expand (Abs(m)) = Abs(map (fn (p,e) =>(p,expand e)) m)
  | expand (Tuple l) = Tuple(map expand l)
  | expand (Quote e) = (reify e)
  | expand (Escape x) = Escape(expand x);
```

Figure 4: Reification and Expansion functions

analyze its argument and, if a match occurs, evaluates the action expression in the appropriate environment. Again, the maybe type mechanism is used to signal success or failure. In this way each clause is encoded by a function of type `value -> value maybe`. The clauses are collected into a `ClosureV` value.

Application is defined by the two functions `apply` and `match`. The `apply` function determines if the function being applied is a closure value or a constructor function. If it is a closure value then the `match` function is used to systematically attempt each clause in the closure and project the resulting value out of the maybe type. If it is a constructor then the appropriate construction is generated.

With the exception of the `Quote` and `Escape` clauses the `eval` function is otherwise straightforward and compositional. The meaning of `Quote` is undefined because it is assumed all `Quote`s were removed by the expansion phase that corresponds to compile-time. The meaning of `Escape` is expressed in terms of two calls to `eval` and one call to `embed`, which is described below. This is the second violation of the principle of compositionality in the interpreter.

```
fun bind (Pvar s,v) f = just(fn x => if x=s then v else f x)
  | bind (Ptuple l,tupleV m) f =
         if (length l) = (length m)
            then (fold (fn (pair,just g) => (bind pair g) | (_,nothing) => nothing)
                       (zip l m) (just f))
            else nothing
  | bind (Pcon(s,p),constructionV(t,v)) f = if s=t then bind (p,v) f else nothing
  | bind (Pstring s,stringV t) f = if s=t then just f else nothing
  | bind _ f = nothing

fun match [] x = raise no_match
  | match (f::m) x =
         (case (f x) of nothing => match m x | just v => v);

fun apply (closureV(m)) x = match m x
  | apply (constructorV c) x = constructionV(c,x)
  | apply _ _ = raise not_function;

fun eval global env x =
let fun matchcl (p,e) x =
        case (bind (p,x) env) of
          nothing => nothing
        | just env' => just(eval global env' e)
in case x of
  Var s => env s
| App(f,x) => apply (eval global env f) (eval global env x)
| StringC c => stringV c
| Abs(m) => closureV(map matchcl m)
| Tuple l => tupleV (map (eval global env) l)
| Quote e => raise not_possible
| Escape e => eval global env (embed (eval global global e))
end;

fun meaning env x = eval env env (expand x);
```

Figure 5: Eval and core support functions

4.3 The embedding function

The embedding function maps a value representing an expression into an expression. That is, in the meta-language in which we are giving semantics, it has the type `value -> exp`. Mappings of this sort are quite rare in semantics. The function itself is quite straightforward; it is given in Figure 6. When applied to a value of type expression it constructs the expression encoded by that value. Muller calls this function $\mathcal{R}^{-1}$ [10].

In this version of `embed` the embedding function is undefined on the reflection operators `Quote` and `Escape`. This restriction forces the reflection to occur at compile-time; it is the distinguishing feature of CRML-style reflection. If expressions involving `Quote` were generated by `embed`, then the evaluation function would have to be modified to deal with dynamically occurring `Quote`s. If expressions involving `Escape` were in the domain of `embed` then either `embed` would have to invoke `eval` to interpret the escaped expression or `embed` would return an expression that still potentially contained `Escape`s. Both of these changes would make it impossible to use the simple stratification of the semantics given in the next section to convert the interpreter into a denotational semantics. Instead, a tower of interpreters of unbounded depth would be required for the semantics. This would also interfere with the compile-time type checking strategy described in Section 5.

4.4 From the interpreter to the denotational semantics

The interpreter fails to be a valid denotational definition for just two reasons: (1) the definition of `reify` is not compositional, and (2) the definition of `eval` is not compositional. This section stratifies the language to solve these two problems.

Both of the offending cases have the same basic structure. The function being defined is applied twice, first to a constituent subexpression, and then to an expression of apparently unbounded complexity. The inner application is compositional; it is valid in a denotational semantics. The outer application is not. The stratification is made by carefully characterizing the domain and range of the function being defined so that the outer application may be replaced by a previously defined function. The case of `eval` is considered first.

In `eval` the offending clause is:

```
| Escape e => eval global env (embed (eval global global e))
```

Since it is known that e has been expanded and reified, the inner `eval` will only be applied to quote-free expressions. Inspection of `eval` and `embed` show that they preserve quote-freeness. Furthermore, the output of `embed` will not have any escapes. So the input to the outer `eval` will be strictly in the core language, which is reflection free.

Thus, to make `eval` compositional, first define `evalCore` on the reflection-free core language in the manner in which `eval` is currently defined. This is a compositional definition. Then modify the definition of the meaning of `Escape` by replacing the outer `eval` with an `evalCore`:

```
| Escape e => evalCore global env (embed (eval global global e))
```

```
fun embedp (constructionV("Pvar", stringV s)) = Pvar s
  | embedp (constructionV("Ptuple", L)) = Ptuple(embedpL L)
  | embedp (constructionV("Pcon", tupleV[stringV c,p])) = Pcon(c,embedp p)
  | embedp (constructionV("Pstring", stringV s)) = Pstring s
  | embedp _ = raise no_embedding
and embedpL (constructionV("Nil",tupleV [])) = []
  | embedpL (constructionV("Cons",tupleV[x,y])) = (embedp x) :: (embedpL y)
  | embedpL _ = raise no_embedding;

fun embed (constructionV("Var", stringV s)) = Var s
  | embed (constructionV("App", tupleV[f,c])) = App(embed f,embed c)
  | embed (constructionV("StringC", stringV s)) = StringC s
  | embed (constructionV("Abs", ms)) = Abs(embedpairL ms)
  | embed (constructionV("Tuple",L)) = Tuple(embedL L)
  | embed (constructionV("Quote",v)) = raise no_embedding
  | embed (constructionV("Escape",v)) = raise no_embedding
  | embed _ = raise no_embedding
and embedL (constructionV("Nil",tupleV [])) = []
  | embedL (constructionV("Cons",tupleV[x,y])) = (embed x) :: (embedL y)
  | embedL _ = raise no_embedding
and embedpairL (constructionV("Nil",tupleV [])) = []
  | embedpairL (constructionV("Cons",tupleV[tupleV[p,e],y])) =
               (embedp p,embed e) :: (embedpairL y)
  | embedpairL _ = raise no_embedding;
```

Figure 6: Embedding functions

The resulting definition of `eval` is now compositional.

The stratification of `reify` is similar. Inspection of `reify` and `expand` shows that the output of both functions is quote-free. Thus again it is possible to define a `reifyCore` function that acts on quote-free expressions. The reification of `Quote` then becomes:

```
| reify (Quote e) = reifyCore (reify e)
```

4.5 Compile-time, Run-time and Reflection

The definition of the meaning function in Figure 5 suggests a natural compile-time/run-time distinction. Expansion occurs at compile-time and evaluation occurs at run-time. However, the definition of evaluation above refers to the `Escape` operator; so reflection has not yet been restricted to compile time.

In this section the definition of compilation is enriched so that `evalCore` can serve as the evaluation function at both compile-time and run-time. This requires that an embedding

```
fun reflect env x =
case x of
  Var s => Var s
| App(f,x) => App(reflect env f, reflect env x)
| StringC c => StringC c
| Abs(m) => Abs(map (fn (p,e) => (p,reflect env e)) m)
| Tuple l => Tuple (map (reflect env) l)
| Quote e => raise not_possible
| Escape e => (embed (evalCore env e));

fun meaning env x = evalCore env (reflect env (expand x));
```

Figure 7: Reflection function and its use in the meaning function

phase be introduced at compile-time to interpret the `Escape` operators. We call this embedding *reflection*. The interpretation of the `Escape`s requires some form of evaluation. Thus, to achieve compile-time reflection, some evaluation will occur at compile time. The definition of `reflect` is given in Figure 7.

This is a radical change to the meaning of compile-time, since it is changing the compilation function from a total function to a partial one. The critical property of compilation that is preserved, however, is that all type errors are detected at compile time. Thus CRML inherits from both cultures: like other languages with reflection compilation is partial, but, true to the ML philosophy, compiled code will not exhibit run-time type errors. Type checking issues are discussed below in the next section. Based on our experience with an implementation of compile-time reflection in this manner, we believe it is a worthwhile extension to the notion of compile-time.

5 Typing CRML

The semantics gives a meaning to compile-time reflective programs, but it doesn't present rules to determine which programs it applies to. This is traditionally the role of a type system. This section explores some of the issues related to typing CRML.

The core language of CRML is Standard ML, with its well known Hindley-Milner type system[9, 8]. In the current implementation CRML type checking is achieved by invoking ML type checking at those points where the semantics refers to `evalCore`. All invocations of the type checker occur at compile-time; thus any program accepted by the compiler is type-correct in the sense that nothing goes "wrong" at run-time[7].

This is a very strange notion of type checking, however. First, in full CRML, which can express divergent meta-programs, type checking and compilation are not guaranteed to terminate. Second, there is not a direct type system for meta-programs; only for the object-programs that they produce. That is, there is not a simple set of rules for abstracting the computational invariants of the meta-programs that determine type correctness. This section explores some of the issues that arise in attempts to give CRML a more traditional type system.

The construction of a natural type checker for CRML must give types to the reflection operators. Giving a type to `Quote` is straightforward: If an expression can be parsed in the current environment then the quoted value has type `exp`. Similarly, the domain of applicability of `Escape` is those expressions of type `exp` that, when evaluated in the appropriate environment, produce a term typeable in the current environment. The appropriate environment is the environment of the corresponding `Quote` or the top level environment if there is no corresponding `Quote` operator.

This scheme for typing `Escape` is unsatisfactory for two related reasons: it appeals to a semantic equality and it is not clear in what environment (or family of environments) the equality is to be interpreted. To see how fundamental these defects are, recall the map-n example from Section 3. In that example, for every value of `n` the expression `‘(mapN n)` has a different type. This suggests that the rule be recast as: Let π be the type assignment of the appropriate environment; for every π-compatible environment, ρ, there exists a term e' and type τ such that $e =_\rho \langle\langle e' \rangle\rangle$ and $\pi \vdash e' : \tau$.

The situation is even more complex when the body of the function defining `mapN` is examined. Consider the information necessary to reason about the type correctness of the first clause in the declaration of `mapNfun`. It is critical that `nconses`, `nheads` and `ntails` are all derived from `iota 1 n`. It is also critical that `number "x" n` and `number "xs" m` are always distinct. Finally, it is necessary to capture exactly how `nconses` builds an environment binding all identifiers in `nheads` and `ntails`. All three of these properties appear to be non-trivial to deduce and require induction to prove.

It is easy to form a reduction argument showing that the set of terms accepted by the current type checking strategy of CRML is not recursive. It is more difficult to answer the question ”Is there a useful recursive type system for a CRML subset”? Based on our experience to date, we are not optimistic about finding a useful CRML subset with a decidable type system. Type checking meta-programs is proving correctness of programs in a Turing-complete notation. We will continue to look at type systems and correctness proofs that describe meta-programs; at the very least we desire a theory of the correctness of meta-programs.

6 Extensions to CRML

This semantic exercise revealed several issues not fully resolved in the current CRML implementation. These issues revolve around the environment in which escaped expressions are reflected. The current implementation (and semantics) reflects residual escaped expressions (i.e. those that survive expansion) in the global environment existing at the lexical point where the global declaration containing the expression resides. This treatment of escape is not uniform. Escapes eliminated by expansion are interpreted in the lexical scope of their associated quote. Residual escapes are interpreted in a global environment.

A more satisfactory solution would support reflection in an explicit, arbitrary environment.

This requires a mechanism for capturing lexical environments and passing them around as first class objects, in much the same manner that *call/cc* captures continuations.

If a language were to provide lexical environment capture to support compile-time reflection, one could imagine an even richer language where expressions could not only be reflected in an arbitrary environment but also evaluated in an arbitrary environment. Such a language would provide precisely the tools necessary to elegantly solve the variable capture problems implicit in many macro systems [3]. We are currently investigating the possibility of building a second version of CRML based upon this model.

The modifications to the interpreter/semantics necessary to make this change to the language definition are given in the appendix.

7 Relationship to Other Work

Current models of reflection are built upon an infinite tower of interpreters each interpreted by the levels below it[5, 11, 4, 2]. By limiting the tower to 2 levels the stratification is particularly simple. This of course means that we cannot give semantics to reflective programs involving more than two levels. These are precisely the programs that cannot be typechecked at compile-time. This is the tradeoff between the flexibility of general reflection for benefits of strong typing.

It is straightforward to extend the interpreter to the infinite tower. The embedding function requires modification to handle quoted and escaped expressions as follows:

```
fun embed ... = ...
  | embed (constructionV("Quote",v)) = reify (Quote(embed v))
  | embed (constructionV("Escape",v)) = embed (eval env (embed v))
```

Unfortunately both `reify` and `eval` are no longer compositional. To stratify this semantics requires an indexed family of functions corresponding to the infinite tower of reflective languages.

8 Conclusions

CRML represents a new breed of typed reflective languages. It builds on two very rich traditions that were previously thought incompatible. The tensions between these traditions have required the careful balancing of several tradeoffs, most notably limiting reflection to compile time and making the compiler and type checker partial functions.

Like many evolving programming languages, CRML is moving through a life cycle of think, implement, experiment, and then think again. This paper is a product of reflection on experience with an implementation, and is the basis of the next cycle of implementation and experimentation.

CRML exists, and all of the semantic functions used in this paper can be executed as Standard ML programs. Copies of CRML and these programs may be obtained by anonymous ftp from the directory `/pub/pacsoft` on `cse.ogi.edu`.

Acknowledgements

We would like to thank Dick Kieburtz and Jeff Lewis for many helpful discussions about the design of CRML. Leonidas Fegaras and Jef Bell provided helpful comments on earlier drafts of this paper.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML compiler, August 1987. Distributed as documentation with the compiler.

[2] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 327–341. ACM Press, 1988.

[3] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[4] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 348–355. ACM, August 1984.

[5] D. P. Friedman and M. Wand. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming*, pages 298–307. ACM, August 1986.

[6] James Hook, Richard Kieburtz, and Tim Sheard. Generating programs by reflection. Technical Report 92-015, Department of Computer Science and Engineering, Oregon Graduate Institute, July 1992.

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

[8] Robin Milner and Mads Tofte. *Commentary on the Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[10] Robert Muller. M-LISP: Its natural semantics and equational logic. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 234–242. ACM Press, June 1991.

[11] Jim des Rivieres and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Proceedings of the 1984 Lisp and Functional Programming Conference*. ACM, 1984.

[12] Tim Sheard. Type parametric programming. Technical Report 93-018, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1993.

[13] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.

[14] Timothy Sheard. Guide to using CRML: Compile-time Reflective ML. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1992.

[15] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.

A First Class Environments

To extend the system developed in Section 4 to support explicit environment capture defined in Section 6 enrich the `exp` type by adding two new constructors for explicit evaluation and lexical capture, and by making escaped expressions contain an explicit lexical environment. Enrich the domain of `values` by adding an environment value constructor.

```
datatype exp = ...
|  Capture of string * exp
|  Eval of exp * (string -> value)
|  Escape of exp * (string -> value)
and value = ...
|  enV of string -> value;
```

Reification and expansion need to be extended to handle the new constructors and to handle reification of escaped expressions differently since escapes now include an explicit environment.

```
fun reify  ...
  | reify (Eval _) = raise not_reifiable
  | reify (Capture(s,e)) = App(Var "Capture",Tuple[StringC s,reify e])
  | reify (Escape(x,e)) = Eval(expand x,e)
and expand ...
  | expand (Eval(x,e)) = Eval(expand x,e)
  | expand (Capture(s,e)) = Capture(s,expand e)
  | expand (Escape(x,e)) = Escape(expand x,e);
```

The `evalCore` function is extended to evaluate the two new constructors:

```
fun evalCore env x =
let fun matchcl (p,e) x = ...
in case x of ...
```

```
| Capture(s,e) => evalCore (fn x => if x=s then enV env else env x) e
| Eval(x,e) => evalCore e x
| Quote e => raise not_possible
| Escape e => raise not_possible
end;
```

The embed function is extended to embed values into expressions built by the new constructors:

```
fun embed  ... = ...
  | embed (constructionV("Eval",tupleV[x,enV e])) = Eval(embed x,e)
  | embed (constructionV("Capture",tupleV[stringV s,x])) = Capture(s,embed x)
```

The reflection operator must remove embedded escapes in the new constructors, and reflect escapes in their explicit environments. Since reflection is done in explicit environments, the reflect operation no longer needs to carry the global environment around as a parameter.

```
fun reflect x =
case x of ...
| Eval(x,e) => Eval(reflect x,e)
| Capture(s,e) => Capture(s,reflect e)
| Quote e => raise not_possible
| Escape(x,env) => (embed (evalCore env x));
```

Finally the meaning an expression can be found by expansion and reification, followed by reflection, followed by evaluation.

```
fun meaning env x = evalCore env (reflect (expand x));
```