

RML – A New Language and Implementation for Natural Semantics

Mikael Pettersson

Email: mpe@ida.liu.se

Abstract

RML is a programming language intended for the implementation of Natural Semantics specifications. The basic procedural elements are *relations*: many-to-many mappings defined by a number of *axioms* or *inference rules*. It has control flow, logical variables and (explicit) unification as in Prolog; from ML it borrows a polymorphic type system, data structures, and pattern matching; a facility for separately-compileable modules also exists. A simple prototype compiler, based on translating RML to Continuation-Passing Style and then to C, has been implemented. Benchmarks indicate that this compiler generates code that is *several orders of magnitude* faster than Typol, and two times faster than standard Prolog compilers.

Partially supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

This is a slightly revised version of a paper presented at the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP'94), Madrid, September 14-16, 1994, published in Springer-Verlag LNCS 844, pages 117-131.

1 Introduction

Natural Semantics, a successor to Structural Operational Semantics, has emerged as a popular tool for specifying type systems, dynamic semantics (interpretation), and compilation [18, 9, 10, 13]. Until recently, only one implementation of the formalism existed: the Typol language within the Centaur system [7]. However, we felt that this implementation was inadequate, both for performance reasons, and because it was too tightly integrated in the Centaur environment.

RML, our Relational Meta-Language, is intended as an alternative to Typol when performance or the ability to build stand-alone applications is important. RML achieves its performance through a combination of language features and compilation strategy.

1.1 Paper Overview

Following this overview, section 1.2 summarizes the implementation strategy. Section 2 presents the basic design of the RML language. Then a Continuation-Passing Style semantics for RML's control flow constructs follows in section 3, together with a discussion on operational properties and optimizations. Sections 4 and 5 describe the bulk of the compilation process, followed by a larger example (section 6) and a summary of the current state of the implementation (section 7). Benchmarks comparing the prototype RML2C compiler first with Typol/Centaur, and then with Quintus and SICStus Prolog are presented next (section 8). Finally, section 9 discusses our immediate future plans, followed by a summary in section 10.

1.2 Implementation Strategy

1. A Continuation-Passing Style denotational semantics for the language is constructed. Certain operational properties are identified.
2. This semantics is trivially turned into a compiler to a CPS representation.
3. Optimizations are applied to the CPS form.
4. Higher-level features (closures, structured data, memory allocation, parameter passing) are made explicit by a translation to a low-level *Code* form.
5. Finally, the Code form is unparsed as C code.

While this strategy seems natural to us, logic programmers may wonder why no mention is made either of translation to Prolog or of using the WAM.

1.2.1 Why not Prolog?

Translating Natural Semantics to Prolog has been done before; it is exactly what the Typol implementation in the Centaur system does. However, such a translation necessarily loses important information. RML is a statically strongly typed language with parametric polymorphism, while Prolog is dynamically typed. RML uses pattern-matching as its primary data inspection mechanism, while Prolog uses unification. Our benchmarks support the hypothesis that a specialized compiler is likely to perform significantly better than would a Prolog compiler on a translated specification.

1.2.2 Why not WAM?

While the WAM's runtime model is fine (section 3.1 will arrive at a very similar one), we see two problems with the programming model:

1. It is too low-level, being an imperative machine with global registers and side-effects. High-level optimizations on WAM code would be difficult.

2. It is too high-level. The instructions are often very complex, with many implicit operations. Making hidden operations *explicit* enables more opportunities for optimizations, as noted by Van Roy [22].

Finally, Attali’s Attribute Grammar approach [4, 5] deserves comment. The observation here is that there exists a particular form of attribute grammars for which efficient evaluators can be mechanically constructed. Then sufficient conditions for Typol specifications to be equivalent to this attribute grammar form are identified. While this does allow *some* Typol specifications to be given more efficient implementations, it is not a general compilation strategy. Any specification outside the designated subset will be implemented in the old way.

2 RML Language Overview

From a language design point, RML is an amalgamation of ideas from Prolog, Standard ML, and (of course) Natural Semantics. The formal definition of RML is described in [17].

From Natural Semantics it has an inference rule/axiom-style of defining relations. The textual layout of rules is a deliberate attempt to be close to conventional notation for Natural Semantics. Individual rules may be named.

From Prolog it has a deterministic execution strategy (top-down, left-right) with backtracking, negation-as-failure ($\backslash+$), logical variables, and explicit unification. The “cut” is not included.

From Standard ML it has a static strong type system with parametric polymorphism, type inference, and **datatype** declarations for user-defined sum-of-products types. The primary mechanism by which relations dispatch on incoming arguments, and calling relations destructure returned values, is pattern-matching. Types, expressions and patterns follow SML syntax where applicable.

Contrary to Prolog, there is a syntactic separation of arguments and results. However, since a variable pattern matches anything, an uninstantiated logical variable in an argument can still be instantiated by explicit unification.

In SML all functions take one argument and produce one result; RML relations are many-to-many.

Finally, there is a system for constructing separately-compilable modules. A module’s interface specifies what types and relations it exports. Other modules’ interfaces may be imported both in interfaces and module bodies.

An example defining a simple expression language is shown in Figure 1.

3 Semantics for RML Control Flow

This section presents a Continuation-Passing Style (CPS) denotational semantics for the RML control flow constructs. This gives a background to later discussions regarding operational properties and the translation to CPS.

Since RML’s control flow is essentially identical to that of Prolog, and the use of continuations for modeling it is well-known [15], it is only briefly discussed here. To simplify the presentation, a subset of RML (RML0) is used, which only retains the basic control-flow constructs, viz. procedure calls, sequencing, disjunction, backtracking, and negation, and the notion of “current” state. See Figure 2 for the CPS semantics for RML0.

There are two kinds of continuations: *success* and *failure* continuations. Success continuations are used for sequencing. As can be seen in the equation for conjunction ($\mathcal{G} \llbracket g_1 \text{ and } g_2 \rrbracket$), the first conjunct is called with an augmented success continuation that knows how to continue with the second conjunct. When a goal or procedure succeeds, it calls its success continuation to “continue.”

Failure continuations are used for backtracking. A success continuation is to be called with the current failure continuation as an argument. Upon failure (not shown, but imagine a primitive procedure that might fail), the then-current failure continuation is called. At a disjunction, the first disjunct is called with an augmented failure continuation that knows how to retry with the

```

module Eval
interface
  datatype Exp      = CONST of int | ADD of Exp * Exp
                    | VAR of string | LET of string * Exp * Exp

  type Env         = (string * int) list

  relation eval(Env, Exp) => int
end

relation lookup =      (* find the most recent binding of an identifier *)
  rule   id' = id
        -----
        "lookup.1"
        lookup((id, val)::_, id') => val

  rule   \+ id' = id & lookup(env, id') => val
        -----
        "lookup.2"
        lookup((id, _)::env, id') => val
end

relation eval =      (* evaluate an expression given an environment *)
  axiom  eval(_, CONST n) => n

  rule   lookup(env, id) => n
        -----
        eval(env, VAR id) => n

  rule   eval(env, e1) => n1 & eval(env, e2) => n2 &
        int_add(n1, n2) => n3
        -----
        eval(env, ADD(e1, e2)) => n3

  rule   eval(env, rhs) => n &
        eval((lhs, n)::env, body) => n'      (* eval body in new env *)
        -----
        eval(env, LET(lhs, rhs, body)) => n'
end

```

Figure 1: RML specification for a simple expression evaluator

Abstract syntax	
$p \in Proc$	procedure names
$g \in Goal \longrightarrow g_1 \text{ and } g_2 \mid g_1 \text{ or } g_2 \mid \text{not } g \mid p$	goals
$d \in Dec \longrightarrow p = g; d \mid (\text{empty})$	procedure declarations
$Prog \longrightarrow d$	programs
Semantic domains	
$l \in Loc$ (unspecified)	locations
$x \in EVal$ (unspecified)	expressible values
$SVal = EVal + \{unbound, unallocated\}$	storable values
$s \in State = Loc \rightarrow SVal$	states
$m \in Marker = State$	state markers
$Answer = \{Yes, No\} \perp$	answers
$fc \in FCont = State \rightarrow Answer$	failure continuations
$sc \in SCont = FCont \rightarrow State \rightarrow Answer$	success continuations
$pe \in PEnv = Proc \rightarrow PVal$	procedure environments
$PVal = PEnv \rightarrow FCont \rightarrow SCont \rightarrow State \rightarrow Answer$	procedure values
Auxiliaries	
$pe_0 : PEnv$	[unspecified set of primitive procedures]
$s_0 : State$	[initial state]
$s_0 = \lambda l. unallocated$	
$new : State \rightarrow (Loc \times State)$	
$new\ s = \langle l, s' \rangle$ where $s\ l = unallocated \wedge s' = s[l \mapsto unbound]$	
$bind : State \rightarrow Loc \rightarrow EVal \rightarrow State$	
$bind\ s\ l\ x =$ if $s\ l = unbound$ then $s[l \mapsto x]$ else –	
$marker : State \rightarrow Marker$	
$marker\ s = s$	
$restore : Marker \rightarrow State \rightarrow State$	
$restore\ m\ s = m$	
Semantic functions	
$\mathcal{G} : Goal \rightarrow PEnv \rightarrow FCont \rightarrow SCont \rightarrow State \rightarrow Answer$	
$\mathcal{G} \llbracket g_1 \text{ and } g_2 \rrbracket pe\ fc\ sc\ s = \mathcal{G} \llbracket g_1 \rrbracket pe\ fc\ \{\lambda fc'. \lambda s'. \mathcal{G} \llbracket g_2 \rrbracket pe\ fc'\ sc\ s'\} s$	
$\mathcal{G} \llbracket g_1 \text{ or } g_2 \rrbracket pe\ fc\ sc\ s =$ $\quad \text{let } m = marker\ s$ $\quad \text{in } \mathcal{G} \llbracket g_1 \rrbracket pe\ \{\lambda s'. \mathcal{G} \llbracket g_2 \rrbracket pe\ fc\ sc\ (restore\ m\ s')\} sc\ s$	
$\mathcal{G} \llbracket \text{not } g \rrbracket pe\ fc\ sc\ s =$ $\quad \text{let } m = marker\ s$ $\quad \text{in } \mathcal{G} \llbracket g \rrbracket pe\ \{\lambda s'. sc\ fc\ (restore\ m\ s')\} \{\lambda fc'. \lambda s'. fc\ s'\} s$	
$\mathcal{G} \llbracket p \rrbracket pe\ fc\ sc\ s = pe\ p\ pe\ fc\ sc\ s$	
$\mathcal{D} : Dec \rightarrow PEnv \rightarrow PEnv$	
$\mathcal{D} \llbracket p = g; d \rrbracket pe = \mathcal{D} \llbracket d \rrbracket (pe[p \mapsto \lambda pe. \lambda fc. \lambda sc. \lambda s. \mathcal{G} \llbracket g \rrbracket pe\ fc\ sc\ s])$	
$\mathcal{D} \llbracket \] \rrbracket pe = pe$	
$\mathcal{E}\mathcal{X}\mathcal{E}\mathcal{C} : Prog \rightarrow Answer$	
$\mathcal{E}\mathcal{X}\mathcal{E}\mathcal{C} \llbracket d \rrbracket = \text{let } pe = \mathcal{D} \llbracket d \rrbracket pe_0 \text{ in } pe \llbracket main \rrbracket pe\ \{\lambda s. No\} \{\lambda fc. \lambda s. Yes\} s_0$	

Figure 2: Denotational semantics of RML0

second disjunct. (See $\mathcal{G} \llbracket g_1 \text{ or } g_2 \rrbracket$.) Also note that a failure continuation records the current state when it is created, and restores this state when it is applied.

The negation of a goal is achieved by trying to prove the goal using a failure continuation that restores the original state before succeeding, and a success continuation that immediately fails using the original failure continuation.

3.1 Operational Properties of RML0

Although RML0 only included *some* parts of full RML, there are some useful operational properties that can be identified.

3.1.1 Strict Evaluation.

As a consequence of the continuation-passing style used for \mathcal{G} , all argument expressions in the right-hand sides are in weak head normal form. This implies that strict evaluation (call-by-value) can be used.

3.1.2 Implicit Global Procedure Environment.

Once \mathcal{D} has iterated through a program's declarations and augmented pe_0 with bindings for user-defined procedures, there is only one pe that gets pushed around in the semantic equations. Hence, in compiled code we can view pe as a global constant, and the pe arguments can be omitted. (Assuming that pe_0 has no `assert/retract`.)

3.1.3 Stack-Allocation of Continuations.

Continuations are λ -abstractions, and normally such values are represented as dynamically allocated closures. Here we describe why RML's continuations may be allocated on a global stack. (A precondition is that continuations do not "escape", i.e. there is no `call/cc`.)

Since the specification is in continuation-passing style, at any given point there is either *exactly* one redex, or the program has reduced to its final answer. Let the first reduction occur at time 1, the second at time 2, and so on. A continuation's *birth-time* is the time at which the reduction occurred that created the continuation. A continuation c_1 is *older than* (*younger than*) a continuation c_2 if c_1 's birth-time is strictly less (greater or equal) than c_2 's birth-time.

First note that failure continuations do not themselves take continuation arguments; hence, when a particular $FCont$ is applied, all younger continuations currently in existence are inaccessible.

Success continuations take failure continuations as arguments. Thus, both continuations in existence when the $SCont$ was created, and continuations reachable via the argument $FCont$, are reachable. However, suppose a success continuation sc is applied to a failure continuation fc , and fc is older than sc . Then, just as when failure continuations are called, all continuations younger than sc are inaccessible.

The upshot of this is that continuations may be represented as stack-allocated records. Every time a continuation is created, its representation (a record containing a code pointer and values of free variables) is pushed onto a global stack. When a failure continuation is called, the stack pointer is reset to the position just before its record. When a success continuation is called, if the $FCont$ argument is younger, then the stack pointer is reset to the position of the $FCont$, otherwise it is reset to the position just before the $SCont$'s record.

Note that the WAM manages its control stack just as described here [24, 1].

3.1.4 State Single-Threadedness and Trailing.

The equations dealing with backtracking ($\mathcal{G} \llbracket g_1 \text{ or } g_2 \rrbracket$ and $\mathcal{G} \llbracket \text{not } g \rrbracket$) clearly illustrate that backtracking must restore the state (bindings of logical variables) to the state at the time the failure continuation (a.k.a. choice point) was created.

As currently specified, the *marker* and *restore* primitives are trivial: *marker s* saves the current state, and *restore m s'* recovers the one saved in the marker. This unfortunately means that the state is not *single-threaded*: at any given time there may exist a number of different states. This has the disastrous effect of preventing state modifications to be implemented by side-effects to an implicit global state [19].

However, in the equations for disjunction and negation, the states m (original s) and s' in the expression *restore m s'* are not unrelated. The following holds:

1. $Dom\ m \subseteq Dom\ s'$
new may have been applied a number of times to allocate new locations.
2. $\forall l \in Dom\ m : lookup\ m\ l \neq unbound \Rightarrow lookup\ m\ l = lookup\ s'\ l$
bind may have been called to update some *unbound* locations.

The most important action of *restore* is to unbind those locations that have been bound since the state was saved in the marker. An obvious way of accomplishing this is to log side-effects: every time *bind* binds a location, the location is recorded in a stack known as the *trail*. *marker s* need then just save the current position of the trail, while *restore m s'* unbinds the locations between the current trail position and the one saved in the marker, before resetting the trail position.

Let locations be allocated in some deterministic order, and let the state record its current “next” location. Then it suffices for *marker s* to record the current next location, and for *restore m s'* to reset it, to achieve the effect of resetting the state’s domain.

3.1.5 Reducing Trailing.

The current specification is still sub-optimal, as a comparison with the WAM shows [1]. The difference is that the RML specification (assuming the operational optimizations described above) trails *all* bound locations, whereas the WAM only records those referable from the “current” failure continuation (a.k.a. choice point).

Since *marker* records the current next free location for use by *restore*, it can also store that value in a global *trailing boundary* variable. Then *bind* would only have to trail bindings of locations *previous* to the current trailing boundary, since later locations are discarded automatically as the domain of the state is reset. (Here *previous* and *later* refer to the allocation order of locations.)

With these modifications, the runtime structure of RML has been made very similar to that of the WAM.

4 Compiling RML to CPS

This section describes the use of CPS — Continuation-Passing Style — as *the* intermediate form used by the RML compiler. First some of the attractive properties of CPS are summarized. Then the particular form used for RML is presented, together with a description of some simple local optimizing transformations.

4.1 Properties of CPS

- Since it forms a closed subset of a call-by-value λ -calculus with constants and implicit state, traditional λ -calculus reduction rules remain valid, and new ones are easily derived. Other representations typically do not relate directly to the semantics of the language, and thus require more work of the compiler writer to invent, and prove correct, rewrite rules.
- Evaluation order is explicit and intermediate results are bound to variables, much like in traditional quadruple or RTL representations.
- It will sometimes give better results than “direct”-style representations in the areas of Abstract Interpretation [14] and Partial Evaluation [8].

Abstract syntax		
$i \in Int$		integers
$r \in Real$		reals
$s \in String$		strings
$w \in Word \longrightarrow fixnum\ i \mid structhdr\ i_{len}\ i_{tag}$		integer words
$c \in Const \longrightarrow w \mid r \mid s$		constants
$v \in Var$		variables
$l \in Lit \longrightarrow const\ c \mid struct\ i_{tag}\ l^*$		literals
$t \in TExp \longrightarrow v \mid l \mid \lambda_{fc}().e \mid \lambda_{sc}(v^*, v_{fc}).e$		trivial expressions
$o \in POpr \longrightarrow deref\ t \mid mkref \mid marker \mid unify\ t_1\ t_2$		primitive operations
	$\mid mkstruct\ i_{tag}\ t^* \mid fetch\ i_{off}\ t$	
$e \in Exp \longrightarrow @_{fc}\ t_{fc} \mid @_{sc}\ t_{sc}\ t^*\ t_{fc} \mid @_p\ v_p\ t^*\ t_{fc}\ t_{sc}$		expressions
	$\mid restore\ t\ e \mid let\ v = t\ in\ e \mid let\ v = o\ in\ e \mid switch\ t\ (c : e)^* e_{def}$	
$Prog \longrightarrow (v_p = \lambda_p(v^*, v_{fc}, v_{sc}).e)^+$		programs

Figure 3: Definition of RML’s internal CPS representation

- Other compilers have demonstrated the usefulness of CPS in the context of (semi-)functional languages [20, 12, 11, 3, 2].

4.2 The CPS representation for RML

The CPS representation of RML programs is a straight-forward encoding of the λ -terms occurring in the RML specification; it is defined in Figure 3. A few comments on terminology and notation: *trivial expressions* are those expressions that denote values rather than computations; variables, abstractions, and applications are considered to be *typed*, as indicated by their *fc*, *sc*, or *p* subscripts; abstractions and applications are uncurried. The primitive operators *mkref* and *mkstruct* allocate logical variables and composite structures respectively; *marker* and the *restore* expression correspond directly to the primitives by the same names in the denotational semantics; *deref* and *fetch* are generated to implement pattern matching, as is the *switch* expression; *unify* implements unification and returns a boolean flag indicating failure or success.

4.3 Local Optimizations on CPS

Figure 4 shows the transformations currently used by RML’s CPS optimizer. The rules assume that all bound variables have unique names, and thus that substitution is trivial. This condition is easily maintained. In rule (β_λ) , t_λ means that t is a λ -expression. In rule (β_{prog}) , λ_p summarizes the entire procedure body. The (β_p) and (β_{prog}) rules need reference counts for all procedures. This requires a simple, per-module, reference count analysis. A procedure exported from a module is considered to have an infinite number of uses.

The η rules serve to compact certain forms of continuations that tend to arise as a result of other optimizations. η -reductions often turn the last procedure call within a body into a tail-call (a.k.a. Last Call Optimization). The β rules perform various forms of continuation and procedure inlining, copy and constant propagation, and dead code removal. The β_r rule uses the fact that every failure continuation originally starts with a call to *restore*, so a *restore* just before an $@_{fc}$ is redundant.

5 Compiling CPS to C

The translation from CPS to C is a two-stage process. First, a CPS program is mapped to a *Code* program in which continuation creation, continuation application, parameter passing, heap allocation and structure initialization, and the allocation of constants has been made explicit. The

(η_{fc})	$[[\lambda_{fc}().@_{fc} t_{fc}]]$	$\Rightarrow t_{fc}$	
(η_{sc})	$[[\lambda_{sc}(v^*, v_{fc}).@_{sc} t_{sc} v^* v_{fc}]]$	$\Rightarrow t_{sc}$	
	if none of the formal parameters occurs free in t_{sc}		
(β_{fc})	$[[@_{fc} (\lambda_{fc}().e)]]$	$\Rightarrow e$	
(β_{sc})	$[[@_{sc} (\lambda_{sc}(\langle v_1, \dots, v_n \rangle, v_{fc}).e) \langle t_1, \dots, t_n \rangle t_{fc}]]$	\Rightarrow	
	$[[let v_1 = t_1 in \dots let v_n = t_n in let v_{fc} = t_{fc} in e]]$		
(β_p)	$[[@_p v_p \langle t_1, \dots, t_n \rangle t_{fc} t_{sc}]]$	\Rightarrow	
	$[[let v_1 = t_1 in \dots let v_n = t_n in let v_{fc} = t_{fc} in let v_{sc} = t_{sc} in e]]$		
	if $[[v_p = \lambda_p(\langle v_1, \dots, v_n \rangle, v_{fc}, v_{sc}).e] \in Prog]$ and v_p is used only once		
(β_r)	$[[restore t (@_{fc} t_{fc}]]]$	$\Rightarrow [[@_{fc} t_{fc}]]]$	
(β_α)	$[[let v_1 = v_2 in e]]$	$\Rightarrow e [v_1 \mapsto v_2]$	
(β_l)	$[[let v = l in e]]$	$\Rightarrow e [v \mapsto l]$	
(β_λ)	$[[let v = t_\lambda in e]]$	$\Rightarrow e [v \mapsto t_\lambda]$	
	if v is used at most once in e		
(β_o)	$[[let v = o in e]]$	$\Rightarrow e$	
	if v is unused and o has no serious side-effect (i.e. not <i>unify</i>)		
(β_{sw1})	$[[switch c \langle \dots (c' : e) \dots \rangle e_{def}]]$	$\Rightarrow e$	if $c = c'$
(β_{sw2})	$[[switch c case^* e_{def}]]$	$\Rightarrow e_{def}$	if no <i>case</i> has tag c
(β_{sw3})	$[[switch v \langle \dots (c : e) \dots \rangle e_{def}]]$	\Rightarrow	
	$[[switch v \langle \dots (c : e [v \mapsto c]) \dots \rangle e_{def}]]$		
(β_{Prog})	$[[\langle \dots, v_p = \lambda_p, \dots \rangle]]$	$\Rightarrow [[\langle \dots, \dots \rangle]]$	if v_p is unused

Figure 4: Local CPS optimization rules

second step is a simple unparsing of the Code representation to a C program text. See Figure 5 for the definition of the Code representation.

5.1 Translating CPS to Code

In order to reduce dependence on the particular machine encoding used for integers, pointers, and object headers, operations for creating machine words and manipulating pointer tags are made explicit.

For each procedure and continuation body, the compiler first computes the amount of heap storage it will need. (An upper approximation due to the CPS *switch* construct.) A translated body will start by calling the *alloc* primitive to allocate the estimated amount of heap needed. (Hence, garbage collection can only occur at this point, before local variables have been bound.) The compiler keeps track of its current offset in the heap block. A CPS *mkref* or *mkstruct* will become a sequence of *store* instructions to initialize the structure, before a tagged pointer to the first word is constructed to represent the value itself.

Every procedure and continuation body is made into a label that is defined by the translation of the body.

Before a call to a procedure or continuation, ordinary arguments are evaluated and placed in the global argument variables ($A\theta$ etc.). Continuation arguments are placed in the SC and FC variables. A procedure or continuation body will, after having performed its initial memory allocation, fetch its parameters from the global argument and continuation variables into local variables.

When a continuation is created, the values of the free variables of its body are pushed onto the stack (relative to the SP variable), and last the continuation's label is pushed. Since only the needed values are pushed, we automatically achieve the *Environment Trimming* optimization of the WAM. To invoke a continuation, the label (at offset 0 relative to the continuation pointer) is fetched and used as the argument to *tailcall*. The invoked continuation will, after calling *alloc*, fetch the values of its free variables from the continuation record into local variables. Then the stack pointer is popped, as described in Section 3.1.3.

Abstract Syntax		
$id \in Ident$		identifiers
$x \in Var \longrightarrow global\ id \mid local\ id$		variables
$lab \in Label$		labels
$w \in Word \longrightarrow fixnum\ i \mid uncoded\ i$		words
	$\mid structhdr\ i_{len}\ i_{tag} \mid unboundhdr$	
$lid \in LitId$		literal “names”
$lref \in LitRef \longrightarrow intval\ w \mid label\ lab$		literal references
	$\mid real\ lid \mid string\ lid \mid struct\ lid$	
$ldef \in LitDef \longrightarrow real\ r \mid string\ s \mid struct\ i_{tag}\ lref^*$		literal definitions
$v \in Value \longrightarrow var\ x \mid rawint\ w \mid literal\ lref \mid fetch\ v$		values
	$\mid offset\ v\ i \mid untagptr\ v \mid tagptr\ v \mid call\ lab\ v^*$	
$c \in Code \longrightarrow tailcall\ v \mid store\ v_{dst}\ v_{src}\ c$		code
	$\mid bind\ x_{opt}\ v\ c \mid switch\ v\ (w : c)^* c_{def}$	
$def \in LabDef \longrightarrow labdef\ (global \mid local)\ lab\ c$		label definitions
$Mod \longrightarrow module\ id\ id^* lab^* (lid, ldef)^* def^*$		modules
Auxiliaries		
$SP, FC, SC, TP, A0, A1, A2, \dots : Var$		global variables
$alloc, deref, isptr, popsc, stringeq, unify, unwind : Label$		

Figure 5: Definition of RML’s internal Code representation

The CPS *marker* operation is translated to bind the current value of the global *TP* (Trail Pointer) to a local variable; *restore* becomes a call to *unwind* with this value as an argument.

5.2 Translating Code to C

Translating Code to C is basically straight-forward: global (local) variables become global (local) C variables, the stack and trail are implemented as global arrays, values become C expressions, code becomes sequences of C statements, and structured literals become statically allocated C data structures.

The only technical problems are how to represent labels, and how to implement tailcalls. Because of tail-recursiveness, the C stack cannot be used for calls or parameter passing. A standard solution is to represent each label as a parameterless function [20, 21]. A function must end by *returning* the address of the next function to call. On top of this sits a *dispatch* loop that calls a function pointer, gets back a new pointer, and then loops to call the new function, ad infinitum. Parameters and results are passed in global variables and/or a simulated stack.

6 A Code Generation Example

Figure 6 shows a simple RML relation, and its CPS and C representations. The RML relation, which maps a tuple of integers to their sum as a one-element list, incorporates most important operational features: data inspection, continuation creation and invocation, calling mechanisms, and storage allocation.

The CPS version illustrates the control flow, and details about data layout. The empty list ($\#(0)$) is a structure with 0 data slots and tag 0, while a cons cell has 2 data slots and tag 1. A plain n -tuple is represented as a structure with n slots and tag 0. Integers and “enumerated” constants have non-boxed representations.

The C version illustrates the use of global variables for parameter passing, how continuation closures are represented as stack-allocated records, explicit heap allocation and structure initialization, and pointer tagging/untagging.

```

relation test = rule int_add(x, y) => z (* RML test case *)
-----
                test((x,y)) => [z]
end

(define (test v102 fc100 sc101)          ; CPS in Scheme-like syntax
  (let ((v103 (rml_deref_box v102)))    ; dereference the argument
    (let ((v104 (rml_fetch 0 v103)))    ; and get its header
      (switch v104
        ((STRUCTHDR 2 0))              ; was it a 2-tuple with tag 0?
         (let ((v106 (rml_fetch 2 v103))) ; v106 = y
             (let ((v105 (rml_fetch 1 v103))) ; v105 = x
               (@pv rml_int_add v105 v106 fc100 ; call int_add with x and y
                 (lambda "sc114" (fc110 v111) ; v111 = z, free vars = sc101
                   (let ((v113 (rml_mkstruct 1 v111 '#(0)))) ; z::nil
                     (@sc sc101 fc110 v113))))))
             (else (@fc fc100))))))      ; unbound argument, so fail

void *test(void) {                      /* Code in C syntax */
  { void *sc101 = rmlSC;
    { void *fc100 = rmlFC;
      { void *v102 = rmlA0;
        { void *v103 = rml_prim_deref_box(v102);
          { void *v104 = FETCH(OFFSET(UNTAGPTR(v103), 0));
            switch( (int)v104 ) {
              case STRUCTHDR(2,0):
                { void *v106 = FETCH(OFFSET(UNTAGPTR(v103), 2));
                  { void *v105 = FETCH(OFFSET(UNTAGPTR(v103), 1));
                    rmlA0 = v105; rmlA1 = v106; rmlFC = fc100;
                    STORE(OFFSET(rmlSP, -1), sc101);          /* store free var */
                    rmlSC = rmlSP = OFFSET(rmlSP, -2);
                    STORE(rmlSP, IMMEDIATE(sclam114));        /* store cont. label */
                    TAILCALL(IMMEDIATE(rml_int_add));}}
                  default: rmlFC = fc100; TAILCALL(FETCH(rmlFC));}}}}}}
static const DEFSTRUCTOLIT(lit0,0);    /* empty list: size 0, tag 0 */
static void *sclam114(void) {
  { void *v116 = rml_prim_alloc(3);      /* 3 words for a cons */
    { void *sc101 = FETCH(OFFSET(rmlSC, 1)); /* fetch free var */
      rml_prim_popsc(2);
      { void *v111 = rmlA0;
        { void *fc110 = rmlFC;
          STORE(OFFSET(v116, 0), IMMEDIATE(STRUCTHDR(2,1))); /*size 2 tag 1*/
          STORE(OFFSET(v116, 1), v111);                       /*car=v111=z*/
          STORE(OFFSET(v116, 2), REFSTRUCTLIT(lit0));         /*cdr=nil*/
          { void *v113 = TAGPTR(OFFSET(v116, 0));
            rmlA0 = v113; rmlFC = fc110; rmlSC = sc101;
            TAILCALL(FETCH(rmlSC));}}}}}}

```

Figure 6: An example of intermediate and generated code

7 Implementation Status

A compiler from RML to portable C code is operational, including the local CPS optimizations. The compiler is written in Standard ML: the lexical analyzer is written in SML-Lex (≈ 160 lines), and the parser in SML-YACC (≈ 500 lines); the rest is ≈ 5200 lines of SML code. The compiler additionally uses a generic pretty-printing engine also written in SML (320 lines).

The generated code is linked with a small runtime system written in C (≈ 1750 lines) which implements standard procedures, dynamic memory management (a very simple copying collector), unification, trailing, trail unwinding, and primitive operations. The trail optimizations described in section 3.1.5 are not yet implemented. The compiler and runtime system currently run on Sun SPARCs under SunOS 4 and SunOS 5, but should be portable to any system for which SML/NJ is available.

8 A Benchmark

In order to evaluate the prototype RML2C compiler, an existing Typol specification was converted to RML, and the two systems benchmarked against each other. The tests were conducted on a Sun SPARCstation 10/41, and the version of Typol was that distributed with Centaur 1.2 [7].

The specification contained evaluation rules for Mini-Freja, a small functional language with *normal-order* (call-by-name) semantics. A Typol version of this semantics was written earlier by two graduate students in our research group (335 lines; 8520 bytes). It was compiled without debugging or coroutining enabled, to intermediate Prolog code (304 lines; 18248 bytes). This code must then be loaded into the Centaur system for execution.

The RML version of the specification (303 lines; 7901 bytes) was compiled to C (2877 lines; 81921 bytes), then assembly (5269 lines; 90402 bytes) and finally object code (18112 bytes). The final executable interpreter (which included parts from the RML runtime library and code to construct the abstract syntax trees of the test cases) totalled 40960 bytes.

The test case was to interpret a Mini-Freja program computing prime numbers using the Sieve of Erathostenes. The Typol code needed 13 seconds to compute the first 3 prime numbers, while the RML code needed 0.03 seconds, i.e. 433 times less. Neither performed any garbage collections. Computing 4 primes required 72 seconds for Typol and 0.04 seconds for RML. Finally, to compute the first 5 primes, the Typol code needed 18 minutes and 50 seconds and garbage collected 3 times. The RML code needed 0.05 seconds, i.e. 22600 times less, while allocating a total of 38672 bytes.

The Mini-Freja specification was then converted to Prolog and compiled using Quintus Prolog (version 3.1.1) and SICStus Prolog (version 2.1 #9). (Both systems compiled to native SPARC machine code.) Re-running the benchmark, now computing 18 primes, took 2.0 seconds for the RML code, 4.5 seconds for Quintus, and 5.0 seconds for SICStus. The times for the Prolog systems do not include start-up initialization. (For Quintus licencing reasons, these tests were run on a Sun 4/470.)

9 Future Work

Much work remains to be done on the compiler. Our immediate plans are to incorporate a real pattern-match compiler [16], and some high-level transformations, primarily aimed at increasing determinacy. Interoperability between RML-generated code and hand-written C code needs to be improved; currently, this is difficult since RML procedures never “return” in any conventional sense.

RML will be used for more specifications, in particular in the areas of type inference, translation, and code interpretation. This will hopefully either confirm the language design choices, or point out where modifications are necessary.

It is also intended to construct two new back ends: one generating SPARC assembly code, and one generating C using Baker’s recent proposal [6]. The SPARC back end would allow faster

compilation and possibly better code, since tailcalls need not be emulated as they are now. Baker’s proposal also seems to offer faster C code from CPS representations.

Finally, we would like to investigate whether certain operational properties could profitably be *specified*, rather than relying on a global analysis that may or may not discover the property in mind. Of particular interest are groundness, determinacy, and single-threadedness conditions. Augmented type system, for instance including subtyping or linearity [23], may be of help here.

10 Summary

A new metalanguage, RML, for the implementation for Natural Semantics specifications has been designed. A prototype compiler generating C, using CPS as its intermediate form, has been implemented. Initial benchmarks indicate that RML2C-compiled code execute much faster than that generated by the Typol implementation, and faster than standard Prolog compilers.

We have shown that CPS is a viable intermediate representation for a logic programming language, and that it can be mapped to efficient code. The absence of `assert`, `retract`, and `call/cc` in RML, and its emphasis of “directed” relations, has helped in this respect.

References

- [1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 1991.
- [2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [3] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proc. POPL'89*, pages 293–302. ACM Press, 1989.
- [4] Isabelle Attali. Compiling TYPOL with attribute grammars. In *Proc. PLILP'88*, LNCS 348, pages 252–272. Springer-Verlag, 1988.
- [5] Isabelle Attali and Jacques Chazarain. Functional evaluation of natural semantics specifications. In *Proc. WAGA'90*, LNCS 461. Springer-Verlag, 1990.
- [6] Henry G. Baker. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. Draft posted to the `comp.lang.scheme.c` newsgroup, February 4, 1994.
- [7] *The Centaur 1.2 Manual*, 1992. Available from INRIA – Sophia Antipolis.
- [8] Charles Consel and Olivier Danvy. For a better support of static data flow. In *Proc. FPCA'91*, LNCS 523, pages 496–519. Springer-Verlag, 1991.
- [9] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proceedings of the 1st Symposium on Logic in Computer Science*, pages 193–205. IEEE, 1986.
- [10] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, LNCS 247, pages 22–39. Springer-Verlag, 1987.
- [11] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Proc. POPL'89*, pages 281–292. ACM Press, 1989.
- [12] David Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: an optimizing compiler for scheme. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM Press, 1986.
- [13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [14] J.A. Muylaert-Filho and G.L. Burn. Continuation passing transformation and abstract interpretation. Research Report DoC 92/21, Imperial College, 1992.
- [15] Tim Nicholson and Norman Foo. A denotational semantics for Prolog. *ACM TOPLAS*, 11(4):650–665, October 1989. Correction in *TOPLAS* 15(1):206–208.
- [16] Mikael Petterson. A term pattern-match compiler inspired by finite automata theory. In *Proc. CC'92*, LNCS 641, pages 258–270. Springer-Verlag, 1992.
- [17] Mikael Petterson. The Definition of RML – Version 1. Forthcoming research report, Dept. of Computer and Information Science, Linköping Univ., 1995.
- [18] Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, September 1981.
- [19] David A. Schmidt. Detecting global variables in denotational specifications. *ACM TOPLAS*, 7(2):299–310, April 1985.
- [20] Guy L. Steele Jr. *Rabbit: a Compiler for Scheme (A Study in Compiler Optimization)*. MIT AI Memo 474, Massachusetts Institute of Technology, 1978.

- [21] David R. Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling Standard ML to C. *ACM LOPLAS*, 1(2):161–177, June 1992.
- [22] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming*. PhD thesis, University of California at Berkeley, 1990.
- [23] Philip Wadler. Is there a use for linear logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM)*, 1991.
- [24] David H.D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, October 1983.