A Relational/Functional Language and Its Compilation into the WAM

Harold Boley

SEKI Report SR-90-05

A Relational/Functional Language and Its Compilation into the WAM

Harold Boley
boley@informatik.uni-kl.de

Deutsches Forschungszentrum für Künstliche Intelligenz
Universität Kaiserslautern

April 1990

Abstract

The first part discusses an amalgamation of relations and functions on the basis of 'valued clauses', as used in RELFUN. These extend Horn clauses by 'foot' premises, specifying the values to be returned. Functions can succeed or fail, enumerate values non-deterministically, return multiple values, and have non-ground arguments and values. Relations act like characteristic functions, permit functionally nested call-by-value arguments, and are definable as higher-order operators. Higher-order clauses are characterized by a structure or a (free) variable in some operator position.

The second part describes the WAM compilation of RELFUN. Multiple-valued functions are transformed to a 'denotative' form, eliminating foots that are active calls. Call-by-value nestings (possibly non-deterministic) are 'flattened'. Higher-order clauses are reduced to 'constant-operator' clauses. Finally, WAM code is generated by extending the use of X-registers and 'put'/'get' instructions: values are put into registers X1, ... just before a clause returns; from there, the caller can get them as arguments, as if loaded by top-level put instructions.

1 Introduction

The applicative (functional) and logic (relational) programming communities are currently investing much effort in the development of independently standardized systems. While acceptable standards for languages like LISP and PROLOG will be of great practical utility, the separate growth of these software worlds also implies increasing duplication of effort. A possible alternative is to integrate the purely functional and relational language kernels, and then to develop a common impure language environment.

Various functional/relational amalgamation approaches have been proposed (a good sample collection is [DL86]). They can be studied with emphasis on *expressive power*, *semantic foundation*, *implementation method*, or *time/space efficiency*. This paper introduces a WAM implementation method [War83] of RELFUN (**rel**ational/**fun**ctional language), an amalgamation based on *valued clauses*. First, however, the expressive power of these

relational/functional clauses is discussed: They permit 'closed' (non-λ) higher-order relational/functional terms while preserving PROLOG's "non-ground (pattern) programming" style. RELFUN's original formal semantics is operational, in the form of a definitional interpreter in pure LISP [Bol86].

We use transformation and compilation techniques to explore the working hypothesis that the efficiency of relational/functional integrations will be able to approach the efficiency of compiled PROLOG. RELFUN's functional features were conceived and implemented in the interpreter without concern for later compilation to WAM instructions. Nevertheless, after some preparatory transformations (mainly *flattening* and *constant-operator reduction*), functions turn out to have a strong affinity to this standard relational implementation method: RELFUN clauses can return values by just putting them, as their last action, into a WAM's "temporary (X-) registers", where a main function call can get them as its actual arguments. Several WAM-like X-register optimizations thus become also possible for functions. Alternatively, flattened, constant-operator-reduced RELFUN clauses can be further transformed to equivalent PROLOG clauses, which may then be compiled for high-efficiency PROLOG machines such as the KCM [BDN+89].

A COMMON LISP prototype of RELFUN is now in experimental use at the University of Kaiserslautern. It is planned to enrich this relational/functional kernel language by a hierarchical type system and augment it by a tailored programming environment. We currently develop a RELFUN package of declarative operations on hypergraphs (semantic nets) [Bol90]. Our principal application area at the German Research Center for AI will be qualitative reasoning in mechanical engineering.

2 The Relational/Functional Language

This is an introductory description of RELFUN. We will distinguish first-order aspects (subsections 2.1 through 2.3) from higher-order aspects (subsection 2.4).

2.1 Amalgamating Relations and Functions

Intuitively, the key idea of the relational/functional amalgamation can be understood in two steps:

1. PROLOG-like relations are augmented to *non-deterministic, non-ground functions.* A RELFUN function extends an n-ary relation by having it deliver a returned value along with variable bindings. Such values may be non-ground terms (e.g. unbound logical variables) and are enumerated using "don't-know" non-determinism (with failure being signaled by the special symbol unknown). Returned values thus correspond to variable values of an (n+1)-ary relation extended by a result argument. However, function calls can return values directly to a main function or relation call in pure LISP's *call-by-value* fashion absent from pure PROLOG.

2. RELFUN functions are specialized again to `true`-*valued relations.* A RELFUN relation can either fail or succeed, as in PROLOG, but while it signals the special symbol unknown on failure, it actually **returns** the truth-value `true` on success. Thus, `true` and unknown can be regarded as the two outcomes of a characteristic function[1].

[1]The interpreted RELFUN version has a third possible outcome, namely the (successful) truth-value `false`.

In summary, because both functions and relations can fail with unknown, their only remaining difference is that on success functions return arbitrary values while relations return `true`. More precisely, although a function may, like a relation, return `true` for certain argument sequences, it can be distinguished from a relation in that it must return a non-`true` value for at least one argument sequence[2]. Therefore, relations and functions have actually become unified to an abstract concept that we will refer to as *operators.*

Note that we deliberately did not call for *determinism* as a defining property of functions but used the more general notion of *non-deterministic functions*, for the following reasons:

- Non-deterministic functions permit the tight amalgamation with relations discussed above (non-ground arguments, invertibility).
- They are often useful for specifying multiple solutions in the fashion of *generators, streams*, or *lazy lists.*
- They can be nested with little overhead via static flattening (cf. 3.3).
- They can be represented semantically as set-valued mappings from a domain to its power set, with each set value representing zero or more enumerable solutions (the empty set represents unknown).
- Determinism specification should be regarded as an optional feature of all operators, relations as well as functions, either *call-time* (using something like PROLOG's `once` predicate) or *definition-time* (using commit operators in clauses or determinism annotations for entire procedures).

2.2 Valued Clauses

Both a relation and a function is defined by a system of *valued clauses*, much like a PROLOG procedure consisting of Horn clauses. The *head* of a valued clause corresponds to the conclusion of a Horn clause. Its *body* and *foot*, both arbitrary-length term conjunctions, correspond to the premises of a Horn clause. Syntactically, the body and foot parts are separated by an ampersand (&) character:

$$head : -body_1, ..., body_B \& foot_1, ..., foot_F. \qquad (1)$$

A *body* premise, like a Horn premise, contributes only success/fail and variable-binding information. A *foot* premise acts like a *body* premise but additionally delivers returned-value information.

Thus, a valued clause acts as if the new separator (&) were replaced by an ordinary conjunction separator (,) but also returns the $v_1 + \ldots + v_F$ values of $foot_1, \ldots, foot_F$ as a non-parenthesized sequence $val_{1,1}, ..., val_{1,v_1}, \ldots, val_{F,1}, ..., val_{F,v_F}$ [3].

[2]In the three-valued version, on success, all calls of a relation must return `true` or `false`, and some call of a function must return a non-true and non-`false` value. Three-valuedness will not be further considered in this paper.

[3]This explicit value sequence should not be confused with the non-deterministic enumeration of values discussed in 2.1. Indeed, the domain mapped to its power set itself consists of sequences built from the base domain of terms: a valued clause can non-deterministically return several value sequences.

All operators can be called as both *body* and *foot* premises, where functional values are taken as mere success signals in the *body* part and a relational value (`true`) is actually returned in the *foot* part.

There are two extreme specializations of the general form (1) of valued clauses.

First, for **valued clauses without any foot (F=0)** the ampersand is omitted, i.e. the top-level syntax of these *footless clauses* coincides with that of PROLOG Horn clauses[4]:

$$head : -body_1, ..., body_B. \tag{2}$$

The complement set of footless clauses is called *footed clauses* (F≥1). Note that both clause sets are **valued** because in RELFUN semantics footless clauses always return `true` on success, hence are equivalent to `true`-footed clauses (F=1):

$$head : -body_1, ..., body_B\&\texttt{true}. \tag{3}$$

A procedure consisting only of footless clauses is thus guaranteed to define a relation. Footless clauses will be further classified like Horn clauses. *Footless rules* have a non-empty body (B≥1, F=0). *Footless facts* are simultaneously bodiless (B=0, F=0), in the below sense, and are written without the ": –" symbol:

$$head. \tag{4}$$

Second, for **valued clauses without any body (B=0)** the "&" is adjoined to ": –", i.e. these *bodiless clauses* can be viewed as unconditional rewrite 'rules' by reading ": –&" as a right arrow (⟶):

$$head : -\&foot_1, ..., foot_F. \tag{5}$$

The complement set of bodiless clauses is called *bodied clauses* (B≥1). Bodiless clauses will again be further classified. In *bodiless rules* the foot is non-empty (B=0, F≥1). *Bodiless facts* are also footless (B=0, F=0), and the ": –&" is omitted as in form (4); they are equivalent to `true`-footed bodiless rules (B=0, F=1):

$$head : -\&\texttt{true}. \tag{6}$$

A procedure consisting only of footed clauses is thus not guaranteed to define a function: it may define a relation all of whose clauses have the form (6) [or, in general, (3)][5].

By combining the above two special cases, valued clauses amalgamate the expressive power of relational and functional programming, permitting conditional, 'narrowing'-like non-ground rewrite rules whose conditions are PROLOG-like goals that can also accumulate partial results.

Regarding the above footless facts and bodiless facts as just *valued facts* (B=0 **and** F=0), we can also employ the complementary notion of *valued rules* (B≥1 **or** F≥1), so that bodied or footed clauses are always bodied or footed rules. In the following, for footed rules we will concentrate on the special case of *single-footed rules* (F=1), but also discuss (in 3.2) the use and implementation of *multiple-footed rules* (F≥2) for *multiple-valued functions.*

[4]In this paper we cannot go into the issue of the empty sequence and of *empty-footed clauses* returning it; they are written by keeping the ampersand: $head : -body_1, ..., body_B\&$.

[5]While these relations are easily recognizable as such, for a clause with an arbitrary foot it cannot be decided in general whether it will return `true` as its only possible value.

2.3 An Example: Refining the palindrome Operator

input argument	output value/signal				output bindings
	palindrome	palinclass	palinzoom	palinlength	
[]	true	even	[]	0	
[a]	true	odd	[a]	1	
[a,b]	unknown	unknown	unknown	unknown	
[b,b]	true	even	[]	2	
[a,d,a]	true	odd	[d]	3	
[a,n,n]	unknown	unknown	unknown	unknown	
[X,n,n]	true	odd	[n]	3	X=n
[n,X,b]	unknown	unknown	unknown	unknown	
[n,X,n]	true	odd	[Center*4]	3	X=Center*4
[s[Y,b],a,Y,X]	true	even	[]	4	X=s[a,b], Y=a
[s[Y,b],a,Y,s[Z,Z]]	unknown	unknown	unknown	unknown	
[[m,Y],a,t[X,X,Y],a,[X,d]]	true	odd	[t[m,m,d]]	5	X=m, Y=d

Table 1: Palindrome variations

The RELFUN amalgamation via valued clauses will now be illustrated by developing a palindrome relation into three functional versions. All four operators shall be defined on lists of arbitrary terms, possibly non-ground. The relational palindrome operator should succeed (with the value true) for palindrome lists and fail (with the signal unknown) for non-palindrome lists. The functional palinclass operator should refine palindrome by differentiating the success cases into palindromes of even and odd lengths, returning the values even and odd, respectively. Table 1 contains I/O samples for these operators.

Note that RELFUN uses "[...]"-brackets not just for lists but also for *denotative* (record) structures such as the list-embedded s[Y,b]. "(...)"-parentheses are used only for *evaluative* (operator) calls such as the palindrome call palindrome([a,d,a]), producing the value true and no bindings. In the table, successful output consists of one value and zero to two bindings; failing output (i.e. the unknown signal) can never be accompanied by any bindings.

All palindrome versions shall build on the usual PROLOG *append* relation. The principal idea is its inverted use to unify the last list element with the first one, and incidentally splitting out the middle list part for recursive calls. Fig. 1 contains the definitions of the palindrome and palinclass operators.

```
palindrome([]).
palindrome([Center]).
palindrome([First-and-Last|Rest]) :-
  append(Middle,[First-and-Last],Rest),
  palindrome(Middle).
```

```
palinclass([]) :-& even.
palinclass([Center]) :-& odd.
palinclass([First-and-Last|Rest]) :-
  append(Middle,[First-and-Last],Rest)&
  palinclass(Middle).
```

```
palinzoom([]) :-& [].
palinzoom([Center]) :-& [Center].
palinzoom([First-and-Last|Rest]) :-
  append(Middle,[First-and-Last],Rest)&
  palinzoom(Middle).
```

```
palinlength([]) :-& 0.
palinlength([Center]) :-& 1.
palinlength([First-and-Last|Rest]) :-
  append(Middle,[First-and-Last],Rest)&
  add1(add1(palinlength(Middle))).
```

Figure 1: Palindrome definitions

The palindrome clauses are footless but could be transcribed to the footed procedure

```
palindrome([]) :-& true.
palindrome([Center]) :-& true.
palindrome([First-and-Last|Rest]) :-
  append(Middle,[First-and-Last],Rest)&
  palindrome(Middle).
```

Here it gets explicit that the empty and singleton palindrome clauses discard their list-type information by both returning true. In palinclass this value becomes refined to the discriminative even and odd values.

The palinclass function can be further refined to a function palinzoom, which returns the listified central element of odd-length palindromes and, the empty list for even-length palindromes. Alternatively, palinclass can be refined to a palinlength function, returning the lengths of palindromes. Sample I/O is shown in Table 1; e.g. the non-ground call palinzoom([a,X,a]) returns the non-ground unit list [Center*4] and binds X to the renamed free variable Center*4. Two definitions in Fig. 1 specify this behavior.

It is possible to represent each (multiple-valued) function by a relation using V additional arguments for binding the V values that were returned by the function. This can be used for a RELFUN-to-PROLOG transformation of (flattened) first-order functions to relations, which makes both PROLOG's model-theoretic semantics and compilation technology indirectly available for first-order RELFUN. For instance, the unary, single-valued palinclass function can be represented by the binary relation palinclass-r:

```
palinclass-r([],even).
palinclass-r([Center],odd).
palinclass-r([First-and-Last|Rest],Class) :-
  append(Middle,[First-and-Last],Rest),
  palinclass-r(Middle,Class).
```

However, if the odd/even information is not often used its inclusion as an additional argument in a user relation palinclass-r would appear rather questionable, while its use as a returned function value of palinclass was quite natural:

1. palinclass can also be used as a unary **predicate** equivalent to palindrome by just ignoring the exact success value in many contexts. The binary palinclass-r relation can only simulate this by 'absorbing' its second argument via an anonymous variable.

2. In the palinclass procedure only the two clauses actually returning additional information are affected by it; the third clause just "passes through" the recursively returned value without static (same source size) or dynamic (same WAM instructions) overhead over the third palindrome clause. In the palinclass-r procedure also the third clause requires a new argument, Class (a "permanent variable" occupying space in the local WAM stack), merely for handing on the recursively bound value[6].

Even if the additional information is employed heavily, the normal use mode is from palindromes to their classes, not vice versa, which is best expressed by an explicit function. Should,

[6]For a detailed comparison of value-returning and value-binding efficiency see [Hei89].

however, the inverse use mode become necessary, the relational version can be called more naturally and efficiently, e.g. by `palinclass-r(Oddpalins,odd)`. The functional version would require RELFUN's generalized `is` primitive (permitting arbitrary, non-arithmetic rhs calls) for inversion by fixing a lhs constant to be unified with the values enumerated via a non-ground rhs call, e.g. by `odd is palinclass(Oddpalins)`. In general, RELFUN therefore offers both functional and relational styles of expression.

2.4 Higher-Order Functions and Relations

Some relational/functional higher-order operations can now be introduced on the basis of the palindrome examples. A more general explanation of higher-order clauses with the syntactic notion of *inconstant-operator clauses* will follow in section 3.4.

The four unary palindrome operations all follow a common recursion scheme that can be abstracted to a higher-order operator `palin[...]`, where `[...]` contains three parameters denoting the value to be produced for the empty list, the function to be applied to a singleton list, and the function to be applied to recursive palindrome values:

```
palin[Emptyval,Singletonfun,Recursionfun]([]) :-& Emptyval.
palin[Emptyval,Singletonfun,Recursionfun]([Center]) :-&
  Singletonfun([Center]).
palin[Emptyval,Singletonfun,Recursionfun]([First-and-Last|Rest]) :-
  append(Middle,[First-and-Last],Rest)&
  Recursionfun(palin[Emptyval,Singletonfun,Recursionfun](Middle)).
```

Suppose we also have defined the generally useful identity, constant, and `twice` (higher-order) functions by

```
id(A) :-& A.
co[C](A) :-& C.
twice[F](A) :-& F(F(A)).
```

Now, instead of first-order operator calls such as `palinclass([a,X,a])` we can parameterize `palin` for higher-order calls such as `palin[even,co[odd],id]([a,X,a])`, which via `palin[even,co[odd],id]([X])` yields `co[odd]([X])`, i.e. returns `odd`.

Alternatively, we can define the original palindrome versions by four fixed `palin` parameterizations returned via function-valued clauses [`palindrome` etc. is used here as the short form of an argumentless call pattern `palindrome()` etc.]:

```
palindrome  :-& palin[true,co[true],id].
palinclass  :-& palin[even,co[odd],id].
palinzoom   :-& palin[[],id,id].
palinlength :-& palin[0,co[1],twice[add1]].
```

Here, e.g. `palinclass([a,X,a])` or `palinclass()([a,X,a])` first evaluates the operator, yielding `palin[even,co[odd],id]([a,X,a])`, which then evaluates as already shown.

The above higher-order clauses employ operators that are **structures** like `twice[F]` or **bound variables** like `F=add1` in `F(F(A))`. In RELFUN it is also possible to employ operators that are **unbound variables** like `Property` in `Property([a,d,a])`. Given the clauses

```
femfirstname([a,d,a]).
langtrademark([a,d,a]).
palindrome([]).     palindrome([Center]).     palindrome(...) :- . . .
```

this request can non-deterministically bind the *relation variable* `Property` three times, by constructively proving $(\exists Property)Property([a,d,a])$:

```
Property([a,d,a]).
~>
true
Property=femfirstname
~>
true
Property=langtrademark
~>
true
Property=palindrome
```

Similarly, with the clauses

```
femprogrammer([a,d,a]) :-& lovelace.
langdeveloper([a,d,a]) :-& [d,o,d].
palinlength([]) :-& 0.   palinlength([Center]) :-& 1.   palinlength(...) :- . . .
```

a *function variable* `Attribute` can be non-deterministically bound three times, using constructive proofs of $(\exists Attribute)(\exists Value)Attribute([a,d,a]) = Value$:

```
Attribute([a,d,a]).
~>
lovelace
Attribute=femprogrammer
~>
[d,o,d]
Attribute=langdeveloper
~>
3
Attribute=palinlength
```

3 Relational/Functional WAM Compilation

After an overview, in the following subsection, we will discuss three transformational compilation phases (subsections 3.2 through 3.4) and the final translation to WAM instructions (subsection 3.5).

3.1 A Compilation Strategy

Implementors of an amalgamated language like RELFUN could either extend functional compilation technology such as SECD and combinator machines toward relations or extend

relational compilation technology such as the WAM toward functions. We chose the latter approach mainly because it nicely supports RELFUN's non-deterministic, non-ground function concept discussed in 2.1.

The main concern, then, was how to extend the WAM [War83] for functional value returning. Our initial approach was the introduction of one additional register, `VALREG`, as the "channel" for returning and fetching single values. WAM instructions for doing this have been implemented in LISP [Hei89]. Our second approach, to be further pursued here, is to use the existing temporary register `X1` for the same purpose. Value returning and fetching can then be done by using existing `put` and `get` instructions. The reason why `VALREG` can be identified with `X1` is that value returning occurs as the last action of clauses, at which time `X1` is no longer needed for argument passing[7]. Using `X1` for both the returned value and the first argument will permit an important optimization of 1-argument nestings: the embedded function can directly put its value into the argument register of the main operation. Furthermore, `X`-register value returning can be generalized naturally to multiple-footed rules, whose V values can be put into the consecutive registers `X1`,...,`X`V, again directly `get`able by a V-argument operation.

The next issue was how to compile the nesting of an arbitrary number of function calls within a main call. However, this problem was already solved in the interpreter by *static flattening* with RELFUN's generalized `is` primitive [Bol86]: a unique variable replaces each nested call, associating with it an `is` primitive conjoined to the left of the main call.

Another preparatory compilation phase, akin to flattening, is *denotative normalization*, which transforms evaluative foots to variables and `is` bodies.

The toughest part is how to compile higher-order operations. We have been following two approaches:

1. A translator can reduce RELFUN's higher-order operators to first-order operands by introducing a new first-order operator `ap`, generalizing the relational `apply` in [War82].
2. The compiler can hash all fixed-arity clauses with operator structures or operator variables to 'collective' procedures, which will be called by corresponding operations and read in their actual operator via the new *operator register* `X0`.

We will elaborate approach 1. here, since this *constant-operator reduction* is much simpler than the direct "higher-order WAM" approach 2., but still can be efficient if the underlying indexing mechanism uses the first two arguments, as done, e.g., in KCM Prolog.

Fig. 2 sequentializes the above phases into the compilation strategy elaborated in the following subsections. The denotative normalizer, static flattener, and constant-operator reducer could be employed in different orders or, indeed, be integrated to a single preprocessing phase, which itself could later be combined with the WAM compiler.

3.2 Multiple-Valued Functions and Denotative Normalization

A term is *denotative* iff it is a constant (e.g. `john`), a variable (e.g. `Who` or `_1`), a structure (e.g. `children[john,Who]`), or a list (e.g. `[Who,[],Who]`). Otherwise it is *evaluative* (e.g.

[7]However, the last action can begin with the first instruction: in the optimization of 'constructor-like' bodiless rules such as `cons(H,T):-&[H|T]`. a `VALREG/X1` separation would allow to "put_list" the cell for `[H|T]` immediately to `VALREG`, unifying the arguments `X1=H` and `X2=T` into it, in order to save one transfer from an auxiliary `X3` to the value register.

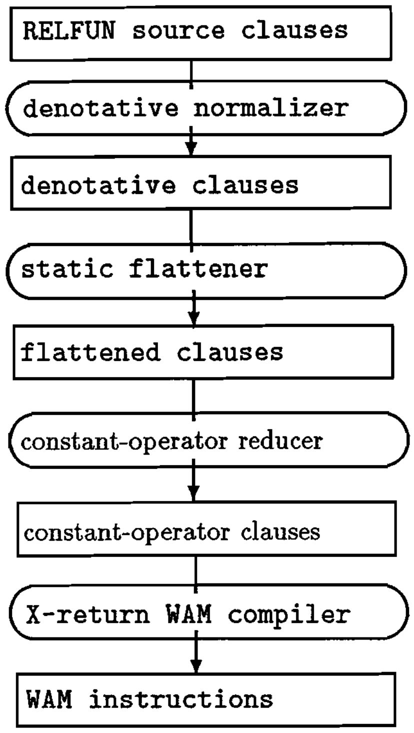

Figure 2: Compilation phases

`children(john,Who)` or `_1 is john`). A footless clause is always (implicitly `true`-) denotative. A footed clause is denotative iff all its foots are denotative terms. Any other clause is evaluative.

Clauses can always be made denotative by replacing evaluative foots by variables. The resulting *denotative normal form* is an intermediate step for compiling multiple-footed rules (see below) and also simplifies the presentation of flattening (see 3.3). A *denotative foot form* is a strong denotative normal form in which footless clauses assume the explicitly `true`-footed form (3) of section 2.2.

Denotative normalization of footed clauses can be defined by a system of two rewrite rule schemata:

$$\ldots :- \cdots \& \ldots, t_0(t_1, \ldots, t_m), \ldots \quad \longrightarrow \quad \ldots :- \cdots \_\mathcal{G}\ \mathrm{is}\ t_0(t_1, \ldots, t_m) \& \ldots, \_\mathcal{G}, \ldots \qquad (1)$$

$$\ldots :- \cdots \& \ldots, p\ \mathrm{is}\ q, \ldots \quad \longrightarrow \quad \ldots :- \cdots p\ \mathrm{is}\ q \& \ldots, p, \ldots \qquad (2)$$

The two $\mathcal{G}$ occurrences in rewrite schema (1) stand for an integer 1, 2, ... generated such that the variable $\_\mathcal{G}$ is not yet used in the clause.

For example, given single-valued `quotient` and `remainder` functions, a 2-valued `divide` function returning the quotient and remainder, in that order, could be defined by the following

bodiless, 2-footed evaluative rule [8]:

```
divide(N,D) :-& quotient(N,D), remainder(N,D).
```

Two applications of the above rewrite schema (1) give us an equivalent 2-bodied, 2-footed denotative rule [9]:

```
divide(N,D) :- _1 is quotient(N,D), _2 is remainder(N,D) & _1, _2.
```

If denotative normalization precedes WAM compilation, the code returning values (in the foot premises) can follow strictly after any (body) code containing `call` instructions, so in the example the `remainder` call cannot overwrite the `quotient` value in `X1`: the foots just put the `quotient` value from `_1` to `X1` and the `remainder` value from `_2` to `X2`.

Since on `call`ing the first foot no other foot values (that could become overwritten) are returned yet, it is sufficient to use a *rest-denotative normal form* for returning multiple values, i.e. only replacing foots after the first one (which thus becomes the last one to be evaluated in the usual left-to-right order, a change that is irrelevant for the pure language considered here). In the example this leads to a 1-bodied, 2-footed evaluative rule:

```
divide(N,D) :- _1 is remainder(N,D) & quotient(N,D), _1.
```

Here, the foot `call` of `quotient` would implicitly put the first `divide` value to `X1`, while the `remainder` value would be explicitly put from `_1` to `X2`.

If a foot is itself a call to a multiple-valued function, e.g. to `divide`, it must be replaced by as many consecutive variables as needed for this call's number of returned values. Also, the corresponding `is` body associates the entire variable sequence with the multiple-valued call, using a "parallel assignment"-like *multiple-variable* `is` primitive $(p_1, ..., p_V)$ `is` q (unifying the value sequence $r_1, ..., r_V$ of q with the variable sequence $p_1, ..., p_V$, as if there was a sequence of single-variable `is` primitives p_1 `is` r_1, ..., p_V `is` r_V)[10].

For example, the `divide` function could be used to define a 5-valued `divtab` function for tabulating divisions in the form *nominator, denominator,* `yield`, *quotient, remainder.* Its bodiless, 4-footed evaluative rule

```
divtab(N,D) :-& N, D, yield, divide(N,D).
```

is (denotatively and rest-denotatively) normalized to the 1-bodied, 5-footed denotative rule

```
divtab(N,D) :- (_1,_2) is divide(N,D) & N, D, yield, _1, _2.
```

[8]For two arguments `11,3` this again returns two values `3,2`; hence it can be self-nested as in `divide(divide(11,3))` $\rightsquigarrow$ `divide(3,2)` $\rightsquigarrow$ `1,1`. In the corresponding WAM instructions, the inner `divide` call would put the constant 3 into the temporary register `X1` and 2 into `X2`; the outer call could directly get these register settings as its actual arguments.

[9]Such a denotative normal form of any multiple-valued function could be further transformed to a relation without transforming subfunction definitions, which exhibits the relation-like I/O symmetry of functions once they permit multiple values:
```
divide-r(N,D,_1,_2) :- _1 is quotient(N,D), _2 is remainder(N,D).
```

[10]A 2-variable `is` would permit a direct, rest-denotative, 2-value-recursive definition of `divide`:
```
divide(N,D) :- lessp(N,D) & 0, N.
divide(N,D) :- greaterorequalp(N,D), (Q,R) is divide(difference(N,D),D) & add1(Q), R.
```

All F-footed rules can be simulated by 1-footed rules using calls to the bodiless, (1-to-F)-footed denotative multi identity, which for fixed F is definable by F clauses just returning their ascending numbers of arguments. Each I-ary source clause (on the left) becomes a 'noop' WAM procedure `multid/`I (on the right):

```
multid(A1) :-& A1.                            multid/1:  proceed
multid(A1,A2) :-& A1,A2.                      multid/2:  proceed
      . . .                                        . . .
multid(A1,...,AF) :-& A1,...,AF.              multid/F:  proceed
```

For example, this is the 1-footed simulation of the non-normalized 2-footed `divide` rule:

```
divide(N,D) :-& multid(quotient(N,D),remainder(N,D)).
```

The `multid` analogue of denotative normalization is flattening, our next issue. In the following subsections we will not further consider the compilation of multiple-footed rules[11].

3.3 Non-Deterministic Nestings and Static Flattening

A term is *flat* iff it is denotative or it is evaluative and has only denotative subterms. Otherwise it is *nested*. A clause is flat iff all its premises are flat. Otherwise it is nested. In particular, a denotative clause is flat iff its body is flat (since it can have only denotative foots, see 3.2).

Clauses can always be flattened by recursively replacing evaluative subterms by variables. In the resulting *flattened clauses* the subterms to be evaluated become `is`-rhs main terms, which simplifies their call-by-value reduction. Since a sequence of evaluative subterms leads to a conjunction of `is` calls, *non-deterministic subterms* can be managed by the WAM's standard backtracking techniques, thus avoiding the direct handling of non-deterministic term nestings.

If we assume that clauses are in denotative foot form (3.2), saving explicit treatment of foot-side nestings, static flattening can be defined by five rewrite rule schemata:

[11]In WAM procedures the "return arity" of multiple-footed clauses should be specified (after a "/" behind the PROLOG-usual "argument arity"): if `f` has one argument and two values, e.g. `f(W):-&b,c.`, and `g` has four arguments and one value, then the 3-argument nesting `g(a,f(b),d)` expands to the 4-argument call `g(a,b,c,d)`, as indicated by the arity specification `g/4/1(a,f/1/2(b),d)` or, omitting 1-return arities, `g/4(a,f/1/2(b),d)`; also, the is call `(K,L) is f(a)` expands to `(K,L) is (b,c)`, as indicated by `(K,L) is f/1/2(a)`. The compiler could then produce a static unknown for nestings or is calls without such arity conformity. Varying argument arity, as in the RELFUN interpreter, or even, return arity, as in our earlier FIT interpreter, would require an additional register, `XMAX`, for passing the actual argument or return arity.

$$\ldots :- \ldots, s_0(s_1, ..., s_{i-1}, t_0(t_1, ..., t_m), s_{i+1}, ..., s_n), \ldots \& \ldots. \quad (1)$$
$$\longrightarrow \quad \ldots :- \ldots, \_G \,\mathbf{is}\, t_0(t_1, ..., t_m), s_0(s_1, ..., s_{i-1}, \_G, s_{i+1}, ..., s_n), \ldots \& \ldots.$$

$$\ldots :- \ldots, s_0(s_1, ..., s_{i-1}, p \,\mathbf{is}\, q, s_{i+1}, ..., s_n), \ldots \& \ldots. \quad (2)$$
$$\longrightarrow \quad \ldots :- \ldots, p \,\mathbf{is}\, q, s_0(s_1, ..., s_{i-1}, p, s_{i+1}, ..., s_n), \ldots \& \ldots.$$

$$\ldots :- \ldots, r \,\mathbf{is}\, s_0(s_1, ..., s_{i-1}, t_0(t_1, ..., t_m), s_{i+1}, ..., s_n), \ldots \& \ldots. \quad (3)$$
$$\longrightarrow \quad \ldots :- \ldots, \_G \,\mathbf{is}\, t_0(t_1, ..., t_m), r \,\mathbf{is}\, s_0(s_1, ..., s_{i-1}, \_G, s_{i+1}, ..., s_n), \ldots \& \ldots.$$

$$\ldots :- \ldots, r \,\mathbf{is}\, s_0(s_1, ..., s_{i-1}, p \,\mathbf{is}\, q, s_{i+1}, ..., s_n), \ldots \& \ldots. \quad (4)$$
$$\longrightarrow \quad \ldots :- \ldots, p \,\mathbf{is}\, q, r \,\mathbf{is}\, s_0(s_1, ..., s_{i-1}, p, s_{i+1}, ..., s_n), \ldots \& \ldots.$$

$$\ldots :- \ldots, r \,\mathbf{is}\, (p \,\mathbf{is}\, q), \ldots \& \ldots. \quad (5)$$
$$\longrightarrow \quad \ldots :- \ldots, p \,\mathbf{is}\, q, r \,\mathbf{is}\, p, \ldots \& \ldots.$$

$\_G$ again stands for a new variable generated on each application of schemata (1) and (3). In schemata (1)-(4) the operator s_0 may itself be an evaluative term because the position i of subterm substitutions is understood to range from 0 to n[12]. The `is` primitives transformed by schemata (3) and (4) may have been generated by applications of schemata (1) and (2), and later, of schemata (3) and (4) themselves.

To illustrate these concepts, we can employ `child` as an undefined binary **functor** in **structures** like `child[P,Q]`, just **denoting** P and Q's children. An embedding of such a denotative term into an evaluative term leaves the main term flat. Thus, the `cares` body of the denotative foot form

```
parental(P) :- cares(P,child[P,Q]) & true.
```

cannot be transformed by the above rewrite system. Instead, a request like `parental(john)` will directly evaluate the request `cares(john,child[john,Q])`, which may succeed using a monolithic fact like `cares(john,child[john,mary])`.

However, we can also employ `child` as a binary **operator** defined by

```
child(john,luzy) :-& ann.
child(john,mary) :-& bob.
```

in **calls** like `child(P,Q)`, **evaluating** to P and Q's children. An embedding of such an evaluative term into another evaluative term makes the main term nested. Thus, the `cares` body of the denotative foot form

```
parental(P) :- cares(P,child(P,Q)) & true.
```

will be flattened by one application of the above schema (1):

[12]This is important for operator evaluation in a higher-order call like `palinclass()([a,X,a])` of 2.4, which flattens to `_1 is palinclass(), _1([a,X,a])`: `_1` = `palin[even,co[odd],id]` is the actual operator value.

```
parental(P) :- _1 is child(P,Q), cares(P,_1) & true.
```

Using the flat body, a request like `parental(john)` initially evaluates the `is` rhs `child(john, Q)`, which may non-deterministically return the solutions `ann` or `bob`. Going through a _1-binding, the first value leads to the request `cares(john,ann)`. This request would fail if we suppose there is only a fact `cares(john,bob)`. But backtracking on the flat conjunction can easily reactivate the `is` rhs. It now returns the second value, which finally leads to the successful request `cares(john,bob)`.

As an example of a deeper nesting consider the second clause of the factorial definition:

```
fac(0) :-& 1.
fac(N) :-& times(N,fac(sub1(N))).
```

Its denotative foot form

```
fac(N) :- _1 is times(N,fac(sub1(N))) & _1.
```

can be flattened by two applications of rewrite schema (3):

```
fac(N) :- _3 is sub1(N), _2 is fac(_3), _1 is times(N,_2) & _1.
```

A flattened form of evaluative bodiless, 1-footed rules, e.g. of the original second `fac` clause, can be obtained easily (disregarding variable names) from their flattened denotative foot form by resubstituting the foot variable, here _1:

```
fac(N) :- _3 is sub1(N), _2 is fac(_3) & times(N,_2).
```

Since on `call`ing the first subterm no other subterm values (that could become overwritten) are returned yet, it is sufficient to use *rest-flattened clauses* in analogy to the *rest-denotative normal form* of 3.2. Here, only the second and later subterms are replaced by flattening variables. For example, this is a 1-footed, list-valued version of `divide`, along with its flattened and rest-flattened forms:

```
divide(N,D) :-& list(quotient(N,D),remainder(N,D)).
divide(N,D) :- _1 is quotient(N,D), _2 is remainder(N,D) & list(_1,_2).
divide(N,D) :- _1 is remainder(N,D) & list(quotient(N,D),_1).
```

In the last form, the `call` to the `quotient` subterm would implicitly put the first `list` argument to `X1`, while the `remainder` value would be explicitly put from _1 to `X2`.

An important, degenerated case is the rest flattening of unary nestings: since there is only one subterm, they need not be flattened at all. The WAM register `X1` thus acts as a fast "communication channel" from the subterm to the main term of such an operator nesting, comparable to the top of the call stack in functional machines. In deeper nestings this leads to chains of consecutive `call`s, each expecting its argument in `X1` and returning its value to `X1`. A simple example is the rest-flattened form of the second factorial clause, which needs only one new variable for a subterm of the binary `times` call, none for the subterm of the recursive unary `fac` call:

```
fac(N) :- _1 is fac(sub1(N)) & times(N,_1).
```

This form is both readable ("... let _1 be factorial of N-1 ...") and efficient ("...; `call sub1` on `X1`=N; `call fac` on `X1`=N-1; put `X1`=(N-1)! to `X2`; ..."). Its actual WAM instructions will be given in 3.5.

3.4 Higher-Order Clauses and Constant-Operator Reduction

A term is *constant-operator* iff it is denotative or it is evaluative and uses a constant as operator and constant-operator subterms as arguments. Otherwise, if some variable, structure, or evaluative term is used in an operator position, the term is *inconstant-operator.* A clause is constant-operator iff its head uses a constant operator[13] and all its premises are constant-operator. Otherwise the clause is inconstant-operator.

The notion of inconstant-operator clauses is a syntactic characterization of a (λ-variableless) subset of the usual concept of higher-order definitions: inconstant-operator clauses, unlike constant-operator clauses, call (bound or unbound) variables, structures, or (values of) evaluative terms in their premises, or are themselves defined with variables or structures as their head operator.

Not included in the inconstant-operator subset are higher-order relations like `transitive` as defined by constant-operator facts like `transitive(ancestor).`, whose second-order characteristics is dependent on `ancestor`'s use as a first-order relation. On the other hand, the following two examples are inconstant-operator clauses:

The variable Rel can be defined as a higher-order relation, 'typed' to be `transitive`, by the variable-head-operator clause

```
Rel(A,C) :- transitive(Rel), Rel(A,B), Rel(B,C).
```

The structure `compose[Fun1,Fun2]` can be defined as a higher-order function by the structure-head-operator clause

```
compose[Fun1,Fun2](A) :-& Fun1(Fun2(A)).
```

In general, RELFUN uses a term representation of operators, where each term (e.g. variable or structure) may play both the role of an operator and of an operand. This very much eases the higher-to-first-order transformation below.

Clauses can always be made constant-operator by introducing a new operator constant, which relegates all (non-primitive) operators to the first operand position. In the resulting *constant-operator-reduced clauses* evaluative terms keep variables, structures, and embedded evaluative terms only as arguments, which greatly simplifies their WAM compilation.

We assume that clauses are in denotative foot form (3.2) and flattened (3.3), saving explicit treatment of foot-side evaluative terms and body-side evaluative subterms; constant-operator reduction can then be defined by three rewrite rule schemata:

$$h_0(h_1,...,h_k) :- \ldots \& \ldots \quad \longrightarrow \quad \mathcal{A}(h_0,h_1,...,h_k) :- \ldots \& \ldots \qquad (1)$$

$$\ldots :- \ldots, t_0(t_1,...,t_m), \ldots \& \ldots \quad \longrightarrow \quad \ldots :- \ldots, \mathcal{A}(t_0,t_1,...,t_m), \ldots \& \ldots \qquad (2)$$

$$\ldots :- \ldots, r \,\mathrm{is}\, t_0(t_1,...,t_m), \ldots \& \ldots \quad \longrightarrow \quad \ldots :- \ldots, r \,\mathrm{is}\, \mathcal{A}(t_0,t_1,...,t_m), \ldots \& \ldots \qquad (3)$$

[13]Clause heads can never be denotative themselves but clause-head arguments must always be denotative.

$\mathcal{A}$ stands for an operator name (conventionally ap) unique for the entire clause set to be transformed. The rewrite schemata are applied only under the following condition: the operator transformed into the first argument must not be $\mathcal{A}$ itself, i.e. $h_0 \neq \mathcal{A}$ in schema (1) and $t_0 \neq \mathcal{A}$ in schemata (2) and (3).[14]

The above flat, footless variable-head-operator clause, transcribed to denotative foot form, yields a constant-operator version by one application of schema (1) and three applications of schema (2):

```
Rel(A,C) :- transitive(Rel), Rel(A,B), Rel(B,C) & true.
ap(Rel,A,C) :- ap(transitive,Rel), ap(Rel,A,B), ap(Rel,B,C) & true.
```

Similarly, the nested, bodiless structure-head-operator clause, transformed to flattened, denotative foot form, yields a constant-operator version by one application of schema (1) and two applications of schema (3):

```
compose[Fun1,Fun2](A) :- _2 is Fun2(A), _1 is Fun1(_2) & _1.
ap(compose[Fun1,Fun2],A) :- _2 is ap(Fun2,A), _1 is ap(Fun1,_2) & _1.
```

In the constant-operator versions the operator variables Rel, Fun1, and Fun2 as well as the operator structure compose[Fun1,Fun2] are all relegated to first-operand positions.

Constant-operator forms not based on denotative normalization or even flattening can be obtained by resubstitution, starting, e.g., from the last composition version:

```
ap(compose[Fun1,Fun2],A) :- _2 is ap(Fun2,A) & ap(Fun1,_2).
ap(compose[Fun1,Fun2],A) :-& ap(Fun1,ap(Fun2,A)).
```

Finally, let us consider a constant-operator version of the palin example of 2.4. It can be obtained by applying the schemata (1)-(3) to the usual preprocessed form and then keeping flattening but resubstituting denotative normalization. In this version of Fig. 3 the higher-order function palin is well prepared for translation to WAM instructions [15].

[14]If the rewrite rules are reformulated in an algorithmic one-pass fashion, as done in the actual implementation, a single ap can be inserted where necessary, without need for any uniqueness or inequality checks.

[15]Constant-operator reduction also affects calls that already used a constant operator such as the append call in Fig. 3. This works since the user definition of such operators is also ap-reduced by the rewrite schemata. Alternatively, we could consider all constant-operator procedures such as the usual append definition as 'primitives', whose (non-clausal) definitions are not accessible to the ap transformation. Instead, for each primitive the constant-operator reducer must generate one new clause, here ap(append,L1,L2,L3) :-& append(L1,L2,L3). It could then also leave constant-operator calls unchanged: the ap reduction of variable-operator calls would handle all higher-order uses of a constant operator (append may be passed as an argument and then be called via this bound variable), and only for such ap calls would a primitive need its newly generated ap clause. The efficiency advantage of this reducer variant increases with the percentage of pre-existing constant-operator clauses. Extended WAM indexing can achieve similar efficiency gains for the original variant.

```
ap(palin[Emptyval,Singletonfun,Recursionfun],[]) :-& Emptyval.
ap(palin[Emptyval,Singletonfun,Recursionfun],[Center]) :-&
  ap(Singletonfun,[Center]).
ap(palin[Emptyval,Singletonfun,Recursionfun],[First-and-Last|Rest]) :-
  ap(append,Middle,[First-and-Last],Rest),
  _2 is ap(palin[Emptyval,Singletonfun,Recursionfun],Middle)&
  ap(Recursionfun,_2).

ap(id,A) :-& A.
ap(co[C],A) :-& C.
ap(twice[F],A) :- _2 is ap(F,A) & ap(F,_2).
```

Figure 3: The ap version of `palin`

3.5 Translation to WAM Instructions

The WAM instructions employed here will not mention the "argument (`A`) registers" of [War83], but use Warren's *temporary (`X`) registers* both for specifying argument passing and temporary processing. Indeed, we will extend the usage of the temporary registers (`X1`, `X2`, ..., `X`V) to a third task: permitting RELFUN clauses to return V values. This generalization makes `X`-register use symmetrical with respect to input arguments, internal auxiliaries, and output values. However, for the single-valued clauses dealt with here, only `X1` will be needed for value returning.

The instructions will be named as in [GLLO85], but with the regular `X`/`Y`-registers counted from 1, not 0 (`X0` is reserved as an operator register, not yet needed here). For readability, *permanent (`Y`) registers* will be referred to by their source variable names, assuming a 'trimmable' name–`Y`I association.

In the 1-footed clauses considered here the single foot can be compiled like a body-side premise (as if the "&" were a ","): we let **all** premises, from left to right, return a value to `X1`, so that the last one (the foot) overwrites `X1` for the final value. Since footless clauses can be rewritten as `true`-footed clauses we do not consider them here, with one exception: for facts the compiler inserts the final instruction `proctrue` as a short form of the two instructions `put_constant true, X1; proceed`.

All kinds of `put` instructions of the form `put_`$\mathcal{K}$ k`, X1` except for $\mathcal{K}$ = `y_variable` can be reinterpreted as value-returning instructions; in particular, `put_x_variable X1, X1` returns an anonymous (free) variable, and `put_structure f/`N`, X1` returns a structure with functor `f/`N, to be filled by N subsequent `unify` instructions. Similarly, all kinds of `get` instructions of the form `get_`$\mathcal{K}$ k`, X1` can be reinterpreted as value-fetching instructions; in particular, with `get_y_variable` and `get_x_variable` the value from `X1` is fetched into permanent and temporary free variables.

RELFUN's `is` calls, p `is` q, can be translated by just translating q and then using an instruction `get_`$\mathcal{K}$ k`, X1` for k-fetching the value that q returned to `X1` (k transcribes the source lhs p and $\mathcal{K}$ represents the corresponding kind of `get`).

Let us look at the transition from 'relational' instructions for a fact to corresponding 'functional' instructions for a 1-footed rule, taking the first factorial clause as an example. While

the relational version gets two arguments and just proctrues, the functional version only gets one argument but puts a non-trivial value:

```
fac(0,1).

fac/2: get_constant 0, X1
       get_constant 1, X2
       proctrue

fac(0) :-& 1.

fac/1: get_constant 0, X1
       put_constant 1, X1
       proceed
```

The compilation of functional nestings can always be done using flattening variables and the `is` primitive. For first-argument (incl. unary) nestings a returned value can be left directly in the `X1`-argument of the main term. Both situations can be illustrated with the second factorial clause (cf. 3.3):

```
fac(N) :- _1 is fac(sub1(N)) & times(N,_1).

fac/1: allocate
       get_y_variable N, X1
       call sub1/1, 2
       call fac/1, 2
       get_y_variable _1, X1
       put_y_value N, X1
       put_y_value _1, X2
       deallocate
       execute times/2
```

While the above *first-argument-nesting optimization* requires no value transport at all, a *final-nesting optimization* can at least avoid the use of a **permanent** flattening variable for one non-first position: if $t_0(...)$ is the final evaluative subterm of a call $s_0(..., t_0(...), s_{i+1}, ..., s_n)$, none of the subterms s_{i+1}, ..., s_n can destroy `X`-registers; hence the returned value of $t_0(...)$ can be `put_x_value`d directly from `X1` to the main call's register `X`i. A simple example is the main `times` call of the second factorial clause, whose first argument is denotative but whose second argument is the final evaluative subterm[16] (this "temporary nesting" is made more explicit in the source line by resubstituting its flattening variable):

[16]Semantic properties such as the commutativity of `times` could lead to source-level transformations usable for further WAM optimizations: left-recursive nestings as in `fac(N) :-& times(fac(sub1(N)),N).` can maximally exploit the first-argument-nesting optimization, here rendering the `put_x_value` superfluous if the `put_y_value` is redirected to `X2`.

```
fac(N) :-& times(N,fac(sub1(N))).
```

```
fac/1: allocate
       get_y_variable N, X1
       call sub1/1, 1
       call fac/1, 1
       put_x_value X1, X2
       put_y_value N, X1
       deallocate
       execute times/2
```

Non-deterministic functions pose no extra problems for WAM translation: values can be enumerated by setting and resetting X1 within the usual try/retry/trust instructions. We give the flattened denotative normal form of the parental example in 3.3 together with its WAM instructions, again compiling _1 as a temporary variable:

```
child(john,luzy) :-& ann.
child(john,mary) :-& bob.
parental(P) :- _1 is child(P,Q), cares(P,_1).
cares(john,bob).
```

```
child/2:     try_me_else c2, 2
             get_constant john, X1
             get_constant luzy, X2
             put_constant ann, X1
             proceed
c2:          trust_me_else_fail
             get_constant john, X1
             get_constant mary, X2
             put_constant bob, X1
             proceed

parental/1:  allocate
             get_y_variable P, X1
             put_x_variable X2, X2
             call child/2, 1
             put_x_value X1, X2
             put_y_value P, X1
             deallocate
             execute cares/2

cares/2:     get_constant john, X1
             get_constant bob, X2
             proctrue
```

Finally, constant-operator-reduced forms of higher-order clauses are compiled as non-deterministic ap procedures, one for each arity. Following are optimized WAM instructions for the palin example in Fig. 3 of 3.4; they constitute a single procedure ap/2 because all ap clauses happen to have arity 2:

```
ap/2: try_me_else a2, 2
      get_structure palin/3, X1
      unify_x_variable X1
      get_nil X2
      proceed

a2:   retry_me_else a3
      get_structure palin/3, X1
      unify_x_variable X3
      unify_x_variable X1
      get_list X2
      unify_x_variable X3
      unify_nil
      execute ap/2

a3:   retry_me_else a4
      allocate
      get_structure palin/3, X1
      unify_y_variable Emptyval
      unify_y_variable Singletonfun
      unify_y_variable Recursionfun
      get_list X2
      unify_x_variable X5
      unify_x_variable X4
      put_constant append, X1
      put_y_variable Middle, X2
      put_list X3
      unify_x_value X5
      unify_nil
      call ap/2, 4
      put_structure palin/3, X1
      unify_y_value Emptyval
      unify_y_value Singletonfun
      unify_y_value Recursionfun
      put_unsafe_value Middle, X2
      call ap/2, 1
      put_x_value X1, X2
      put_y_value Recursionfun, X1
      deallocate
      execute ap/2

a4:   retry_me_else a5
      get_constant id, X1
      put_x_value X2, X1
      proceed

a5:   retry_me_else a6
      get_structure co/1, X1
      unify_x_variable X1
      proceed

a6:   trust_me_else_fail
      allocate
      get_structure twice/1, X1
```

```
unify_y_variable F
put_y_value F, X1
call ap/2, 1
put_x_value X1, X2
put_y_value F, X1
deallocate
execute ap/2
```

Of course, in practice, indexing instructions would be used to let the procedures `ap/`*I* directly `switch` on their first arguments, the original operators: former constant-head-operator clauses and structure-head-operator clauses can be efficiently accessed with `switch_on_constant` and `switch_on_structure`. Second-argument `switching` would then achieve, for higher-order clauses, the efficiency of PROLOG's standard first-argument indexing.

4 Conclusions

Besides the PROLOG-like syntax adopted up to this point we use a LISP-like syntax for RELFUN's valued clauses: the multiple-footed, single-footed, and hornish (footless) clause notations are `(ft` *head* $body_1$ ... $body_B$ `&` $foot_1$ ... $foot_F$`)`, `(ft` *head* $body_1$ ... $body_B$ $foot_1$`)`, and `(hn` *head* $body_1$ ... $body_B$`)`, respectively. The terms *head*, $body_I$, and $foot_J$ also use Cambridge-Polish notation: evaluative terms like `child(P,Q)` become `(child _p _q)` and denotative terms like `child[P,Q]` become `'(child _p _q)`. In special cases, clauses can be expressed equivalently, as in `(ft (parental _p) (cares ...) & true)` $\Leftrightarrow$ `(ft (parental _p) (cares ...) true)` $\Leftrightarrow$ `(hn (parental _p) (cares ...))`.

With the exception of the yet unimplemented multiple-footed clauses, this syntax is employed in the present interpreter+compiler/emulator system, running in COMMON LISP[17]. The PROLOG-like syntax is currently being implemented as an alternative pretty printer; it will also become the back-end of a RELFUN-to-PROLOG translator, whose central transformation principle was illustrated in 2.3.

Our general compilation approach stresses optimization of the WAM instructions, not speediness of the compiler. The RELFUN source transformation phases are thus formulated as pure, recursive LISP functions (a version of the constant-operator reducer is even written in RELFUN itself).

The WAM translation phase constitutes a much larger LISP program because it exploits many (register) optimizations already in the DATALOG and DATAFUN (structureless and listless RELFUN) subsets. We have defined *classified RELFUN* as an explicit representation language, intermediate between preprocessed RELFUN procedures and WAM instructions; e.g., the present DATAFUN classifier extends clauses by *permanent/temporary*, *safe/unsafe*, and *first/non-first* declarations for variables, collects premises into *chunks* (in the sense of Debray), specifies the *argument sequence* for goal unification, etc. [Kra90]. The code generator thus has a platform from which it can almost read off the WAM instructions, but it also introduces additional low-level optimizations.

We are experimenting with two LISP-based WAM emulators, one transcribed from Beer, the other adapted from Nystrøm [Hei89]. Using our `X1`-return compiler, only the printing of

[17]The original interpreter [Bol86] used only single-footed clauses `(ft` *head* $body_1$ ... $body_B$ $foot_1$`)` and bodiless, footless clauses or facts `(hn` *head*`)`, leaving out the `ft` and `hn` tags.

returned values had to be added to these PROLOG emulators for obtaining RELFUN emulators. We have confirmed a small emulator-performance increase proceeding from relational to equivalent functional definitions of the `palinclass-r/palinclass` type. In order to quantify the LISP overhead, we have translated parts of the Nystrøm emulator to C, extrapolating a speed-up factor of six for this entire emulator [Els90]. On the basis of these experiences we plan extensions of a high-speed PROLOG machine, the SIEMENS-manufactured KCM, for RELFUN.

References

[BDN+89] H. Benker, M. Dorochevsky, J. Noyé, B. O'Riordan, A. Sexton, and J.C. Syre. The knowledge crunching machine at ECRC: A joint R&D project of a high speed PROLOG system. *ICL Technical Journal*, pages 737–753, November 1989.

[Bol86] Harold Boley. RELFUN: A relational/functional integration with valued clauses. *SIGPLAN Notices*, 21(12):87–98, December 1986.

[Bol90] Harold Boley. Declarative operations on nets. *Computers & Mathematics with Applications*, 1990. Forthcoming.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, 1986.

[Els90] Klaus Elsbernd. Effizienzvergleiche zwischen einer LISP- und C-codierten WAM. Technical Report SWP-90-03, University of Kaiserslautern, Department of Computer Science, June 1990.

[GLLO85] John Gabriel, Tim Lindholm, E.L. Lusk, and R.A. Overbeek. A tutorial on the Warren abstract machine for computational logic. Technical Report ANL-84-84, Argonne National Laboratory, Illinois, June 1985.

[Hei89] Hans-Günther Hein. Adding WAM instructions to support valued clauses for the relational/functional integration language RELFUN. Technical Report SWP-90-02, University of Kaiserslautern, Department of Computer Science, December 1989.

[Kra90] Thomas Krause. Klassifizierte relational/funktionale Klauseln: Eine deklarative Zwischensprache zur Generierung von Register-optimierten WAM-Instruktionen. Technical Report SWP-90-04, University of Kaiserslautern, Department of Computer Science, May 1990.

[War82] David H. D. Warren. Higher-order extensions to PROLOG: are they needed? *Machine Intelligence*, 10:441–454, 1982.

[War83] David H.D. Warren. An abstract PROLOG instruction set. Technical Report 309, SRI International, AI Center, 1983.