Lisp and Symbolic Computation, 2, 51–89 (1989)

Integrating Logic and Functional Programming*

HASSAN AIT-KACI (*hassan@mcc.com*)
ROGER NASR (*nasr@mcc.com*)
Microelectronics and Computer Technology Corporation
Austin, Texas 78759, U.S.A.

Abstract. The intent of this article is twofold: To survey prominent proposals for the integration of logic and functional programming and to present a new paradigm for the same purpose. We categorize current research into four types of approaches, depending on the level at which the proposed integration is achieved. Unlike most current work, our approach is not based on extending unification to general-purpose equation solving. Rather, we propose a computation delaying mechanism called *residuation*. This allows a clear distinction between functional *evaluation* and logical *deduction*. The former is based on the λ-calculus, and the latter on Horn clause resolution. Residuation is built into the unification operation which may then account for β-reduction. In clear contrast with equation-solving approaches, our model supports higher-order function evaluation and efficient compilation of both functional and logic programming expressions, without being plagued by non-deterministic term-rewriting. In addition, residuation lends itself naturally to process synchronization and constrained search. We describe an operational semantics and an implementation of a prototype language called LeFun—Logic, equations, and Functions.

1. Introduction

A surge of research activity has been devoted lately to "integrating logic and functional programming." As usual, arguments ranging from matters of taste or pragmatic performance, to deep theoretical concerns have been put forth, some quietly, some forcefully. We, the authors, do not wish to contribute to the debate. Rather than telling the rest of the world how this ought to be done, or even why it ought to be done at all, we shall abide by a more peaceable mode of describing what we do, why we do it, and how. By no means, however, do we wish to appear "holier than thou!" Indeed, we think that some other proposals have definite elegance, are of practical use, or even achieve high performance. Rather, our answer came to us naturally when we tried to define precisely what *we* wanted, and realized that none of the proposals known to us would answer *all* and *only* our needs. Thus, we shall attempt to motivate our work by first laying out our *desiderata*; then, reviewing some of the prominent proposals known to us, pointing out where each falls short of some of our wishes[1].

Here is the way we organized this article. Section 2 introduces our specific motivation. There, we first state our wishes, then we survey a sample of alternative endeavors[2]. Along the way, we shall point out how our wishes are more or less satisfied, or how some ideas will concur with our proposal. The latter is overviewed

*This article is a revised and extended version of [1].

in Section 2.3. In Section 3 we illustrate some details, operational points of our idea, by means of Le Fun example programs. Section 4 presents Le Fun's unification algorithm, which accounts for dynamic function evaluation. Section 5 gives a state transition semantics for Le Fun. The implementation realizing Le Fun's *modus operandi* is detailed in Section 6.4. Finally, a discussion relating our work to similar approaches closes the main body of this article. Appendix A gives more examples of how Le Fun's execution models constraint propagation.

2. Motivation and background

> (...) the link between a 'higher' (in the sense of 'more complex') and a 'lower' field results neither in a reduction of the first to the second nor in greater heterogeneity of the first, but in mutual assimilation such that the second explains the first, but does so by enriching itself with properties not previously perceived (...)
>
> Jean Piaget, *Main Trends in Interdisciplinary Research.*

2.1. Desideratum

To start with, let us define what we mean by functional programming and logic programming. A better qualifier would be "functional and relational" in the following sense.

- By functional, we understand a (1) *directional*, (2) *deterministic*, and (3) *convergent* flow of information.
- By relational, we understand a (1) *multidirectional*, (2) *non-deterministic*, and (3) *not necessarily convergent* flow of information.

That is (1) functions expect input and return output, whereas relations do not, (2) functions do not fail or backtrack, whereas relations do, and (3) functions must terminate on all legal input, whereas relations may enumerate infinitely many alternative instances of their arguments.

Now that we have defined our terminology, it seems that functional programming is subsumed by relational programming. In a pragmatic sense, this is untrue since the specificity of functional programming allows the *elimination of rather heavy computational overhead.* Importantly, we shall view functional programming as computation in *evaluation* mode (no information guessing is allowed) and relational programming as computation in a *deduction* mode (information guessing is allowed). This is an

essential delineation since it explains why functional programming can easily handle higher-order objects. Indeed, missing arguments cannot be synthesized in functional programming as they can in relational programming by way of unification. Synthesizing higher-order objects is, if not impossible, at best computationally very difficult and expensive [31].

Although it is often the case that application problems can be solved entirely in evaluation or deduction mode, these do not constitute all programming applications. From our programming experience using a logic or functional programming language, we have repeatedly found ourselves in frustrating situations where parts of the problem we have at hand were of a functional nature, and others of a relational nature. Of course, we could always fit those parts into the language, but at the cost of some unnatural and often non-trivial thinking. Thus, having better things to do than to argue with the High Priests of each camp, we came to the natural conclusion that both modes were appropriate in different situations. Many others seem to have as well conceded this, and have thus proposed solutions. Let us review some in light of our wishes.

2.2. A survey

Roughly, there seem to be four main trends: an *embedding* approach; a *syntactic* approach; an *algebraic* approach; and a *higher-order logic* approach.

The embedding approach typically proposes that a particular logic or functional language is all that one needs by showing how one can "easily and elegantly" implement the other camp's approach without losing access to the native powers of the chosen language. The syntactic approach is characterized as accommodating the functionality of the other camp through syntactic sugar. The algebraic approach uses the notion of term rewriting as the operational semantics for functional programming. The higher-order logic approach takes the daring stand of immersing computation in combinatory logic which has the theoretical power of handling functions and relations alike.

2.2.1. The embedding approach. This is the most immediate approach. It consists of implementing X in Y so as to have all the goodies of X without losing Y's. For example, Darlington [16] implements sets (logical predicates) as set expressions in a KRC manner [64], Mellish and Hardy implement Prolog in POP-2 (PopLog) [45], Robinson allows logic quantifiers in Lisp [57], and Felleisen "transliterates" Prolog into Scheme [22].

This approach does not meet our *desideratum* in that it does not really integrate logic and functional programming, insofar as a BASIC implementation of a Lisp interpreter does not integrate BASIC and Lisp. The implementation language is simply the metalanguage of the implemented language and obscures the potential for native computational optimization of the rendered language.

2.2.2. The syntactic approach. Let us take the standard "higher-order" functional definition of *map*, a function which returns the list of f-images of elements of a list.

```
map(f,l) = if l = nil
           then nil
           else cons(f(head(l)), map(f,tail(l)))
```

This example uses standard informal syntax of (sugared) λ-calculus [40]. An equational syntax which may conveniently give the advantage of being dually interpreted as rewriting rules as well as yet another sugaring of the λ-calculus, would be:

```
map(F,[])    = [].
map(F,[H|T]) = [F(H)|map(F,T)].
```

Another example of such a "higher-order" definition expresses the *fold* functional which cumulatively performs a given operation on a list of arguments.

```
fold(Op,Id,[])    = Id.
fold(Op,Id,[H|T]) = Op(H, fold(Op,Id,T)).
```

In these examples, we use the now well-known and convenient syntax of Prolog's notation for lists using square brackets, and capitalized logical variables. This syntax is not hard to understand. However, the reader is likely to fancy more than one semantics, even operational ones, which may compute with this functional specification. Thus, besides the λ-calculus, various combinatory logics with various combinator bases [63, 32], as well as categorical logic [13, 14], and algebraic semantics, whether first-order [39], second-order [12], or ω-order [15, 24, 25], to name just a few, could be used, some being perhaps a bit of an overkill for the purpose of a simple functional evaluator.

One may thus choose to proceed as syntactically as possible, and let such notation as above be the tip of a preprocessing iceberg. Thus and typically, the works that we classify as using this approach choose some Turing-equivalent (operational) semantics, like Natural Deduction (*e.g.*, Prolog [65]), Equational Logic (*e.g.*, TEL [62]), or Applicative Programming [4], and give methods to transform statically a given syntax of terms intended for one computational semantics into one alternate such logical, equational, or functional specification whose realization on the operational engine at hand behaves according to the intended semantics—all in a manner that could be made invisible to the user. Let us look at a couple of these proposals to illustrate our point. Namely, Warren's [65] and Smolka's [61] desugaring of applicative expressions into Prolog and TEL, respectively.

Warren's proposal suggests using Prolog as the operational target for a sugared notation of functional expressions, and Smolka's uses typed equational logic with innermost narrowing[3]. Both methods center around representation of curryed functions—partial applications—in the language of first order terms—*i.e.*, fixed-arity

terms of first-order type. The aim is to provide an operationally explicit trick for making sense out of syntactic terms which do not respect arity constraints or allow functional formal arguments. This is achieved by expressing everything at a meta-level with a binary function *apply* denoting (curryed) function application. No assumption need thus be made on the arity of function symbols—which become 0-ary (constant) symbols in the meta-alphabet, perhaps with the pragmatic exceptions of constructors; *i.e.*, symbols that denote data structures. By the same token, this also transforms variables previously of higher-order sort into first-order variables.

Thus, for Warren, an application of a function to n arguments is seen as an $(n + 1)$-ary predicate whose $(n + 1)$-st argument is to be the result of applying the function to the n first arguments. Applicative function composition is unnested into a sequence of n subsequences of predicate invocations—one subsequence per function argument—followed by one application of the outermost function feeding the n results into the ultimate invocation. For example, such an expression as $f(g(a, b), f(c))$ denotes the result of applying a function called f, of arity at least 2, to two arguments, which are respectively the result of applying a function g, of arity at least 2, to two arguments a and b, and the function f to one argument c. The leaves a, b and c denote indefinite constants. Thus, using its operational semantics with a ternary *apply* predicate such that *apply(X, Y,Z)* means $Z = X(Y)$, the Prolog way of paraphrasing the above denotation would be:

Conjunct	Meaning	
`apply(g,a,X),`	$X = g(a)$	$= \lambda x.g(a,x)$
`apply(X,b,Y),`	$Y = \lambda x.g(a,x)(b)$	$= g(a,b)$
`apply(f,Y,Z),`	$Z = f(g(a,b))$	$= \lambda x.f(g(a,b),x)$
`apply(f,c,U),`	$U = f(c)$	$= \lambda x.f(c, x)$
`apply(Z,U,V),`	$V = \lambda x.f(g(a,b),x)(\lambda x.f(c,x))$	$= f(g(a,b),\lambda x.f(c,x))$

where the denoted value is referenced by the logical variable V. Note that in order to be correctly executable in Prolog, explicit currying of all possible partial applications for f and g must somehow be provided. Namely,

```
apply(f,X,f(X)).
apply(f(X),Y,f(X,Y)).
apply(g,X,g(X)).
apply(g(X),Y,g(X,Y)).
```

Warren's proposal amounts to generating explicitly the necessary additional clauses. Looking back at the *map* example, this is desugared into Prolog as:

```
apply(map,F,map(F)).
apply(map(F),[],[]).
apply(map(F),[H|T],[FH|FT])
:- apply (F,H,FH),
   apply(map(F),T,FT).
```

and the *fold* example is desugared as:

```
apply(fold,Op,fold(Op)).
apply(fold(Op),Id,fold(Op,Id)).
apply(fold(Op,Id),[],Id).
apply(fold(Op,Id),[H|T],Val)
:- apply(Op,H,Oph),
   apply(fold(Op,Id),T,Tval),
   apply(Oph,Tval,Val).
```

In Smolka's TEL programming language (Typed Equational Logic [62]), if f is a defined n-ary function, currying is achieved by introducing n new k-ary constructors for $0 \leqslant k \leqslant n-1$ and new equations for a binary *apply* combinator of the form:

$$apply(f_k(X_1,\ldots,X_k),[Y_1,\ldots,Y_l]) = f_{k+l}(X_1,\ldots,X_k,\ Y_1,\ldots,Y_l)$$

for each k and l such $0 \leqslant k \leqslant l \leqslant k+l \leqslant n$, and where f_n stands for f[4]. At the same time, all partial f-expressions (application of n-ary f to less than n arguments) of the form $f(t_1,\ldots,t_k)$ are desugared into expressions of the form $f_k(s_1,\ldots,s_k)$ where the s_i's are the desugared t_i's.

Again, the trick also takes care of higher-order variables by desugaring all expressions of the form $F(t_1,\ldots,t_k)$ into $apply(F,[s_1,\ldots,s_k])$, thus turning them into first-order variables. Even further, λ-expression of the form $\lambda x.t$, where t contains k free variables $x_1,\ldots,x_k$, is desugared into $f_k(x_1,\ldots,x_k)$, where f is a new symbol defined by the new equation $f(x_1,\ldots,x_k,x) = t$.

This syntactic transformation will take the *map* definition into:

```
map(F,[])    = [].
map(F,[H|T]) = [apply(F,[H])|map(F,T)].

apply(map0,[F])     = map1(F).
apply(map0,[F,L])   = map(F,L).
apply(map1(F),[L]) = map(F,L).
```

and the *fold* definition into:

```
fold(Op,Id,[])   = Id.
fold(Op,Id,[H|T] = apply(Op,[F,fold(Op,Id,T)]).

apply(fold0,[Op])       = fold1(Op).
apply(fold0,[Op,Id])    = fold2(Op,Id).
apply(fold0,[Op,Id,L]) = fold(Op,Id,L).
apply(fold1(Op),[Id])   = fold2(Op,Id).
apply(fold1(Op),[Id,L] = fold(Op,Id,L)
apply(fold2(Op,Id),[L])= fold(Op,Id,L).
```

The syntactic transformation approach does not meet our *desideratum* either. Both Warren's and Smolka's methods are making the assumption that somehow functions will not be synthesized. When *apply* is invoked with its first functional argument uninstantiated, many spurious functions are examined as potential candidates. However, most such calls do not denote anything (*e.g.*, consider the call `apply(F,[0,1,2],[1,2,3])` in the context of Warren's *map* definitions). Such calls will aimlessly wander through all possible currying of all known functions before eventually failing—time and space consuming for a simple failure.

Warren's idea looks less prolific in the number of generated definitions. However, it is not difficult to see that it will pay some run-time price that Smolka's statically explosive method will not. Indeed, Warren's method insists on systematically currying all functions. In particular, if a call has all or some of its arguments present, these will be sequenced into successive unary abstractions. This, naturally, makes run-time function application less efficient since n formal arguments mean n function invocations. Whereas, Smolka's method uses memo-lists of arguments already present at parse-time, and provides for run-time argument memoizing as well, thus generating less run-time calls to the detriment of an explosive number of new definitions, not to mention as many new constructor symbols to garbage-collect.

Of course, one could venture many ways of optimizing the methods, but this would strike as odd since one would then be trying to "fix a fix" rather than the original problem.

2.2.3. The algebraic approach. We put in this category works that are based on equational logic as part of their operational semantics. Such proposals are due to Goguen and Meseguer (EqLog) [28], Hsiang [30], Josephson and Dershowitz (Rite) [35], Fribourg (Slog) [23]. The reason why we do not put Smolka's TEL [62] in this category is that although he expresses the semantics of his language as equational logic, the implementation relies on compiling equations into Horn clauses whose execution by Prolog realizes innermost narrowing. Other more or less operational approaches (*e.g.*, [2, 55, 67]) have been proposed recently that exploit some of the same ideas with roughly the same justifications, some a *priori*, others perhaps as afterthoughts. We shall rapidly gloss over EqLog and Hsiang's approach.

Goguen and Meseguer have stressed *Initiality* as the essence of model-theoretic semantics for programming languages [27, 46]. The idea has great elegance and simplicity, and is a model-theoretic property which guarantees that all well-formed the syntactic constructs denote a precise abstract object in all models. Thus, using first-order terms on a ranked alphabet as syntactic objects for the representation of programs (a Herbrand Universe), one can be guaranteed a unique interpretation (homomorphism) of programs into their abstract denotations, sets of which elements constitute models for well-formed programs. All such program expressions unambiguously denote all and only elements of the model. This is paraphrased by Goguen and Meseguer as *no junk, no confusion* models. The elegance of the theory appears as the initiality property is preserved under equational congruences for finitely presented equational models. That is, adding (finitely many) equations (any finite binary

relation) to the language of term expressions generates a congruence relation where *one* same denotation is now given in the same unambiguous way to (congruence) classes of expressions. Hence, semantics of a program expressed as a finite number of first-order equations is obtained as a simple algebraic quotient.

Operationally, this translates as term-rewriting as a computation procedure (term evaluation), and equation-solving as deduction (equal denotation for two terms). The latter may be given many realizations depending on the various restrictions one may put on the syntax of terms. Narrowing [21, 33, 56], congruence closure [7, 26], Knuth-Bendix completion [18, 19], and all variations thereof induced by miscellaneous E-unification operations [52, 66, 37, 38], are thus all effectively complete enumeration procedures of sets of principal solutions of equations. Divergence is, of course, possible when no solution exists or the complete set of solutions is unbounded.

Understanding the concept of initiality, one must naturally ask whether this nice property can be taken further into model varieties that are not equational. That is, what if program specifications are not longer just equations, but built in richer languages like logics and sub-logics of any order. More precisely, the question may be cast in so many words as, *"Is there a larger (largest?) class of models than first-order equational varieties which can be guaranteed the existence of an initial model?"* A very beautiful discovery by two model-theorists, Mahr and Makowsky [42], provides an answer to this question for first-order varieties[5]. Namely, the Mahr-Makowsky result states that yet another class of models which is guaranteed the initiality property are those models algebraically characterized as being closed under direct products. Another name for these structures which is perhaps more familiar to the reader is *Horn Models*. That is, models of Horn Logic. In other words, if in addition to equations, Horn clauses were also used to specify programs, a very clear model theoretic semantics is obtained for free, together with the natural operational semantics of Horn clause resolution interleaved with any complete equation-solving in lieu of unification.

Thus is EqLog justified and defined: exploiting greater power of initial model semantics as well as being operationally realized by, say, Prolog and Narrowing. Undoubtedly, this has great syntactic and semantic clarity and tidiness. It is however strictly first-order—although surely all the syntactic tricks of Smolka's apply. Also, non-determinism of narrowing introduces yet another source of pragmatic problems, even when completeness of outermost narrowing is used to simplify redex choice at unification time. Namely, functions are no-longer determinate[6], with the granted relative advantage of being invertible. Although, in our view, if one is interested in inverting functions, one should not define them as functions to begin with, but as relations—which Prolog technology may "invert" at will.

Hsiang's approach is also cleverly exploiting an algebraic result of equational logic. It may be paraphrased as follows. Since first-order logic can be cast algebraically into a complete set of ten reductions—a finite presentation of a boolean ring—and since so-called logic programming is defined as some sort of decision procedure in a boolean algebra, one can use the Knuth–Bendix method as a decision-procedure. Specifically, a (logic) program is now seem as an equational specification enriching the

set of complete reductions. Computation is the Knuth–Bendix completion procedure altered such that critical-pair computation is biased from the equational query of the form $q = true$ or $q = false$ to the added set of rules, from which in turn critical pair computation is performed by need toward the complete set of reductions.

Thus, evaluating a query is finding a sequence of equational lemmas, if one exists, and a variable assignment which prove that the logical term q is tautologically equivalent to *true* or *false*. For example, proving a first-order sentence s is done by adding the equation $s' = true$, where s' is the skolemized version of s. If no additional rules are generated by the completion procedure, then the sentence is proven (since s and *true* are shown to be in the same equational congruence class). If more rules are generated, the sentence is false. For the reader with background in Automatic Theorem-Proving, this may be reminiscent of a combination of *set-of-support* resolution (the ten complete reductions) and *linear* resolution (resolving from goal to assertions). Naturally, the process will diverge in the case where the query is unsolvable.

Our *desideratum* is yet again not met by the algebraic approach. Admittedly, the idea is a good one, especially since it constitutes a radical departure from resolution-like logic programming, and is operationally novel. It allows full first-order logic as may be expressed using Hsiang's approach. There are however many hang-ups as far as pragmatic exploitation of the method is concerned. The first being that it is not clear that this is more efficient that Prolog technology. The second being that Knuth–Bendix completion demands a simplification ordering on terms for rule orientation, opening a Pandora's box of many yet unsolved problems, especially when combined with E-unification (*e.g.*, termination of AC-rewriting). Finally, this method being based on first-order equational logic as Goguen–Meseguer's and Smolka's, the same observations apply. At any rate, the method is a very active field of research which deserves attention.

2.2.4. The higher-order logic approach. Naturally, there are also those who do not hesitate to make the jump into hyperspace. Indeed, since the λ-calculus is a common language to combinatory logic and functional programming, and knowing that the key operation in ω-order logic (higher-order unification) is semi-decidable [31, 29] —*i.e.*, not worse than, say, Horn clause resolution—why not just use these tools, which are available, to construct a (relatively) complete operational calculus. Such is the approach of Nadathur's and Miller's [47, 49], who explore programming language design based on higher-order logics.

Strange as it seems, this approach is not any more complex than first-order equation solving methods. In fact, the two methods are operationally homomorphic as we shall see. That is, we can transform a higher-order equation into a first-order equation modulo a first-order theory. We summarize the trick next. This trick was suggested to us by Dale Miller.

Let us consider the λ-calculus, augmented with logical variables of any finite order —*i.e.*, first-order (element) variables, second-order (function, or set, or predicate) variables, third-order (functional, or set of sets, or meta-predicate) variables, *etc.* We

have as usual, constants (of any order), functional abstractions, and applications[7].

Now, higher-order unification is simply solving equations in this language, modulo λ-conversion, a binary relation on λ-terms which is the composition of three other relations:

(α) *alphabetical renaming*—which considers two terms equivalent up to a renaming of their λ-bound identifiers.

(β) *functional application*—which relates a λ-term of the form $(\lambda x.e)(e')$ to the term $e[x \leftarrow e']$, *i.e.*, the term e in which the term e' is substituted for all free occurrences of x in e.

(η) *extensionality*—which asserts that a functional term f is equivalent to any term of the form $\lambda x.f(x)$, where x does not occur free in f. This relation expresses that functions which are pointwise equal are equal.

Now, considering the (non λ-bound) variables as constants, we can translate these λ-terms into combinations (binary application trees) of *combinators*—for example, using the standard S, K, I basis (although any complete combinator basis would do [5, 63]).

Recall that,

$$I = \lambda x.x$$

$$K = \lambda x.\lambda y.x$$

$$S = \lambda f.\lambda g.\lambda x.f(x)(g(x))$$

Rewriting this using well known syntactic sugar yields the following equations:

$$I(x) = x$$

$$K(x, y) = x$$

$$S(f, g, x) = f(x)(g(x))$$

Thus, the *SKI*-calculus has three reduction rules. Of course, this looks almost first-order, and invites a confusion. Indeed, it looks as if I is a unary operator, K is a binary operator, and S is a ternary operator—except that the third right-hand side would be syntactically ill-formed.

A better trick is to consider the following first-order ranked operator signature $\Sigma = \{@_2, s_0, k_0, i_0\}$, where subscripts denote arity, @ stands for *apply*, and s, k, i are three distinguished constants; together with the following reduction rules:

$$@(i, X) \rightarrow X$$

$$@(@(k, X), Y) \rightarrow X$$

$$@(@(@(s,F),G),X) \rightarrow @(@(F,X),@(G,X))$$

What happens now if one plays the game of narrowing modulo this set of reductions? That is, can equations be solved after a syntactic sweetener translated your favorite syntax of the λ-calculus into the above first-order combinations? The answer is simple: *Yes—higher-order unification.*

Of course, with these combinators and rules alone innermost narrowing is not possible. However, it is possible with the following set of combinators $\{S, S', S'', K, K', I\}$ together with the translation rules:

$$kx \rightarrow k'x$$

$$Sf \rightarrow S'f$$

$$S'fg \rightarrow S''fg$$

and the reduction rules:

$$Ix \rightarrow x$$

$$K'sy \rightarrow x$$

$$S''fgx \rightarrow (fx)(gx)$$

And now, consider $\Sigma = \{@_2, i_0, k_0, s_0, k'_1, s'_1, s''_2\}$ and the new first-order rewrite rules:

$$@(i,X) \rightarrow X$$

$$@(k,X) \rightarrow k'(X)$$

$$@(k'(X),Y) \rightarrow X$$

$$@(s,F) \rightarrow s'(F)$$

$$@(s'(F),G) \rightarrow s''(F,G)$$

$$@(s''(F,G),X) \rightarrow @(@(F,X),@(G,X))$$

can be cast immediately in the following (innermost narrowing) Prolog program:

```
apply(i,X,X).
apply(k,X,kl(X)).
apply(kl(X),_,X).
```

```
apply(s,F,s1(F)).
apply(s1(F),G,s2(F,G)).
apply(s2(F,G),X,Y)
   :- apply(F,X,FX),
      apply(G,X,GX),
      apply(FX,GX,Y).
```

Clearly, any complete combinator basis will do. However, pragmatic trade-offs of time *versus* space may arise, as more combinators mean more equations, which in turn mean more nondeterminism. Nevertheless, more combinators, if adequately chosen, also mean more compact programs, and thus less narrowing work.

Yet another game one can play with the above is constructing a "higher-order" EqLog where higher-order terms are translated as combinations. Interleaving narrowing modulo combinator reduction with Prolog hence yields a higher-order Prolog.

Our *desideratum* is overmet by the higher-order logic approach. Namely, using higher-order unification (*i.e.*, synthesizing program expressions) is going beyond simple needs for programming. This is clearly too powerful and expensive a tool.

2.3. Overview of our approach

We now introduce a relational and functional programming language called Le Fun where first-order terms are generalized by the inclusion of *applicative expressions* as defined by Landin [40] (atoms, abstractions, and applications) augmented with logical variables. The purpose is to allow *interpreted* functional expressions to participate as *bona fide* arguments in logical expressions.

A unification algorithm generalized along these lines must consider unificands for which success or failure cannot be decided in a local context (*e.g.*, function applications may not be ready for reduction while expression components are still uninstantiated.) We propose to handle such situations by delaying unification until the operands are ready. That is, until further variable instantiations make it possible to reduce unificands containing applicative expressions. In essence, such a unification may be seen as a residual equation which will have to be verified, as opposed to solved, in order to confirm eventual success—whence the name *residuation*. If verified, a residuation is simply discarded; if failing, it triggers chronological backtracking at the latest instantiation point which allowed its evaluation.

Although primarily motivated as an experiment in integrating logic programming (Horn clause resolution) and functional programming (as in λ-calculus style functional reduction), this residuation principle can also be generalized beyond just unification (*i.e.*, syntactic equality) to encompass any syntactical decisions which can be made pending further instantiation. In particular, ground-decidable predicates like arithmetic inequality, or syntactic inequality (forbidding physical identity) can be *implicitly* handled by residuation.

A remarkable corollary of this is that such unclean patches as Prolog's *is* evaluation

predicate are no longer needed, yielding a truly more declarative operational semantics. In that sense, the programmer can describe her problem as a combination of function definitions and Horn clauses where the order in which conjuncts are verified for a given query is completely independent of the order in which they are specified. This frees the programmer from cumbersome explicit control annotations [9, 59, 54]. Indeed, residuation exhibits a flavor of asynchronous computation model that may prove amenable to efficient parallel implementation [51, 41].

3. Le Fun examples

Exposing our ideas is better done by illustrating key points of the residuation principle, giving very simple examples focusing attention away from details.

3.1. Unifying reducible expressions

SLD-resolution on which pure Prolog is based is not a complete deduction system for Horn logic because its depth-first control strategy may diverge although finite solutions exist. In addition, Prolog implementations are also incomplete because of built-in arithmetic. Of course, it is possible to manipulate numbers through a first-order axiomatization of arithmetic. However, performance of a "real-life" programming language forbids this. Thus, arithmetic is built into Prolog as a primitive system. Of course, this is done at the expense of completeness since numbers are thus not synthesized by unification. As a result, a goal literal involving arithmetic variables may not be proven by Prolog, even if those variables were to be provided by proving a subsequent goal. This is why arithmetic expressions cannot be nested in literals other than the *is* predicate, a special one whose operation will force evaluation of such expressions, and whose success depends on its having no uninstantiated variables in its second argument.

We give two simple examples on how this poses no problem to Le Fun.

3.1.1. Simple case. Consider the set of Horn clauses:

```
q(X,Y,Z) :- p(X,Y,Z,Z), pick(X,Y).

p(X,Y,X + Y,X*Y).
p(X,Y,X + Y,(X*Y)-14).

pick(3,5).
pick(2,2).
pick(4,6).
```

and the following query:

```
?- q(A,B,C).
```

From the resolvent $q(A, B, C)$, one step of resolution yields as next goal to establish $p(A, B, C)$. Now, trying to prove the goal using the first of the two p assertions is contingent on solving the equation $A + B = A * B$. Naturally, using Peano's axioms to solve this is out of the question. At this point, Prolog would fail, regardless of the fact that the next goal in the resolvent, $pick(A, B)$ may provide instantiations for its variables which may verify that equation. Our solution is to stay open-minded and proceed with the computation as in the case of success, remembering however that eventual success of proving this resolvent must insist that the equation be verified. As it turns out in this case, the first choice for $pick(A, B)$ does not verify it, since $3 + 5 \neq 3 * 5$. However, the next choice instantiates both A and B to 2, and thus verifies the equation, confirming that success is at hand.

To emphasize the fact that such an equation as $A + B = A * B$ is a left over granule of computation, we call it a *residual equation* or *equational residuation—E-residuation, for short.* We also coin the verb *"to residuate"* to describe the action of leaving some computation for later. We shall soon see that there are other kinds of residuations. Those variables whose instantiation is awaited by some residuations are called *residuation variables* (RV). Thus, an E-residuation may be seen as an *equational closure* —by analogy to a lexical closure—consisting of two functional expressions and a list of RV's.

There is a special type of E-residuation which arises from equations involving an uninstantiated variable on one hand, and a not yet reducible function expression on the other hand (*e.g.*, $X = Y + 1$). Clearly, these will never cause failure of a proof, since they are equations in solved form. Nevertheless, they may be reduced further pending instantiations of their RV's. Hence, these are called *solved residuations* or *S-residuations.* Unless explicitly specified otherwise, "E-residuation" will mean "equational residuations which are not S-residuations."

Going back to our example, if one were interested in further solutions to the original query, one could force backtracking at this point and thus, computation would go back eventually before the point of residuation. The alternative proof of the goal $p(A, B, C, C)$ would then create another residuation; namely, $A + B = (A * B) - 14$. Again, one can check that this equation will be eventually verified by $A = 4$ and $B = 6$.

One may observe that a possible realization of the residuation principle would be to accumulate all residual equations along a depth-first walk of the *and/or* proof tree until a leaf is reached; then, instantiate all E-residuations with the substitution at hand; and succeed if and only if they are all verified. Clearly, this would be far more expensive than using any relevant instantiations *as they materialize.* This is very reminiscent of the process of asynchronous backpatching used in one-pass compilers to resolve forward references.

3.1.2. Trickier case. Since instantiations of variables may be non-ground (*i.e.*, may contain variables), residuations *mutate.* To see this, consider the following example:

```
q(Z) :- p(X,Y,Z), X = V - W, Y = V + W, pick(V,W).

p(A,B,A*B).

pick(9,3).
```

together with the query:

```
?- q(Ans).
```

The goal literal $p(X, Y, Ans)$ creates the S-residuation $Ans = X * Y$. This S-residuation has RV's X and Y. Next, the literal $X = V - W$ instantiates X and creates a new S-residuation. But, since X is an RV to some residuation, rather than proceeding as is, it makes better sense to substitute X into that residuation and eliminate the new S-residuation. This leaves us with the *mutated* residuation $Ans = (V - W) * Y$. This mutation process has thus altered the RV set of the first residuation from $\{X, Y\}$ to $\{V, W, Y\}$.

As computation proceeds, another S-residuation instantiates Y, another RV, and thus triggers another mutation of the original residuation into $Ans = (V - W) * (V + W)$, leaving it with the new RV set $\{V, W\}$.

Finally, as $pick(9, 3)$ instantiates V to 9 and W to 3, the residuation is left with an empty RV set, triggering evaluation, and releasing the residuation, and yielding final solution $Ans = 72$.

3.2. Residuating ground-decidable predicates

Equations are not the only computations which may be residuated. A goal literal whose decision is entailed by grounding its arguments would gain to be potentially suspended, expecting its arguments to become ground. More precisely, and n-ary predicate symbol p is said to be *ground-decidable* if it is immediately possible to decide, given any *ground terms* $t_1, \ldots, t_n$, whether the literal $p(t_1, \ldots, t_n)$ holds true or not. Examples of such predicates are the so-called "built-in" predicates of Prolog such as $\neq (t_1 \neq t_2$ succeeds if and only if t_1 and t_2 do not unify) and arithmetic *comparisons* ($<, \leqslant, >, \geqslant$). Thus, such predicates residuate in Le Fun. These are called *I-residuations.*

Consider, for example,

```
q(X,Y,Z) :- p(X,Y,Z), X < Y, Y < Z, pick(X,Y).

p(X,Y,X*Y).

pick(3,9).
```

with the query,

```
?- q(A,B,C).
```

Understanding this example is left as exercise.

3.3. Higher-order expressions

The last example illustrates how higher-order functional expressions and automatic currying are handled implicitly. Consider,

```
sq(X) = X*X.

twice(F,X) = F(F(X)).

valid_op(twice).

p(1).

pick(lambda(X,X)).

q(Val) :- G = F(X), Val = G(1), valid_op(F),
          pick(X), p(sq(Val)).
```

with the query,

```
?- q(Ans).
```

The first goal literal $G = F(X)$ creates an S-residuation with the RV set $\{F, X\}$. Note that the "higher-order" variable F poses no problem since no attempt is made to solve. Proceeding, a new S-residuation is obtained as $Ans = F(X)(1)$. One step further, F is instantiated to the *twice* function. Thus, this mutates the previous S-residuation to $Ans = twice(X)(1)$. Next, X becomes the identity function, thus releasing the residuation and instantiating Ans to 1. Finally, the equation $sq(1) = 1$ is immediately verified, yielding success.

4. Interpreted unification

In this section we present a basic formal syntax of terms which are a blend of λ-calculus terms and first-order constructor terms. We also define substitutions for these terms. Then, we describe a non-deterministic unification algorithm for these terms which accounts for β-reduction. This algorithm is presented as a set of solution-preserving transformations on a set of equations *à la* Martelli–Montanari [43].

4.1. Terms and substitutions

Let $\{\Sigma_n\}_{n\geqslant 0}$ be an indexed family of mutually disjoint sets of *constructor* symbols of arity n. Let $\Sigma = \cup_{n\geqslant 0}\Sigma_n$ be the set of all constructors. Let V be a countably infinite set of *variables*. By convention, variables will be capitalized not to confuse them with constructor constants in Σ_0.

Let $\mathcal{T}$ be the set of *terms* defined as the smallest set such that:

- if $X\in V$ then $X\in\mathcal{T}$;
- if $a\in\Sigma_0$ then $a\in\mathcal{T}$;
- if $c\in\Sigma_n$ and $t_i\in\mathcal{T}$, $(1\leqslant i\leqslant n)$ then $c(t_1,\ldots,t_n)\in\mathcal{T}$;
- if $X\in V$ and $t\in\mathcal{T}$ then $\lambda X.t\in\mathcal{T}$;
- if $t_1\in\mathcal{T}$ and $t_2\in\mathcal{T}$ then $t_1(t_2)\in\mathcal{T}$.

We shall denote by $t[t']$ the term t with a distinguished occurrence of subterm t'. *Free* and *bound* occurrences of variables in terms are defined exactly as usual as in the λ-calculus. We shall call **var**(t) the set of *all* variables (free or bound) in a term t. The expression $t[X\leftarrow t']$ stands for the term resulting from simultaneously substituting all *free* occurrences of the variable X in t by t'.

There are three basic *reduction* relations defined on terms: α, β, and η. They are defined thus:

- α-reduction: $\lambda X.t \succ_\alpha \lambda Y.t[X\leftarrow Y]$, if Y does not occur free in t;
- β-reduction: $(\lambda X.t)(t') \succ_\beta t[X\leftarrow t']$;
- η-reduction: $\lambda X.t(X) \succ_\eta t$, if X does not occur free in t.

Combined reduction relations (*e.g.*, $\alpha\beta, \beta\eta$) denote the *union* of the basic reductions composing them (*e.g.*, $t_1 \succ_{\alpha\beta} t_2$ iff $t_1 \succ_\alpha t_2$ *or* $t_1 \succ_\beta t_2$).

We shall use the greek letter ξ as a generic parameter standing for any (basic or combined) reduction relation. The following relations are derived from any reduction.

- ξ-*conversion*: $t[t_1] \to_\xi t[t_2]$ if $t_1 \succ_\xi t_2$;
- *symmetric* ξ-conversion: $t_1 \leftrightarrow_\xi t_2$ iff $t_1 \to_\xi t_2$ *or* $t_2 \to_\xi t_1$;
- *reflexive transitive* ξ-conversion: $t_1 \overset{*}{\longrightarrow}_\xi t_2$;
- ξ-*equivalence*: $t_1 \approx_\xi t_2$ iff $t_1 \overset{*}{\longleftrightarrow}_\xi t_2$, where $\overset{*}{\longleftrightarrow}_\xi$ is the reflexive and transitive closure of $\longleftrightarrow_\xi$.

Given a term, there are in general many ways in which one can apply a reduction. A term which cannot be (ξ-)reduced anymore is said to be in (ξ-)*normal form*. As in the pure (untyped) λ-calculus, there may be infinite sequences of reductions. However, there is no ambiguity among terminating reductions thanks to the following very important and well-known result (See [3]):

Theorem 1 (Church-Rosser Property) *If a term t has a β-normal form it is unique up to $\alpha\eta$ conversion.*

Given a term t, we shall denote by $t\downarrow_\beta$ its β-normal form, if it exists. We shall write $t\downarrow_\beta = \perp$ if it has no β-normal form (*i.e.*, if all β-reductions from t do not terminate).

A *substitution* σ is a function from V to τ such that the set $\{X \in V | X \neq \sigma(X)\}$ is finite. This set is called the *domain* of σ and denoted by $\mathbf{dom}(\sigma)$. The set $\{t \varepsilon \mathcal{T} | t = \sigma(X), X \in \mathbf{dom}(t)\}$ is called the *range* of σ and denoted by $\mathbf{ran}(\sigma)$. Such a substitution is also written as a set such as $\sigma = \{t_i/X_i\}_{i=1}^n$ where $\mathbf{dom}(\sigma) = \{X_i\}_{i=1}^n$ and $\sigma(X_i) = t_i$ for $i = 1$ to n. A substitution σ is in (ξ-)normal form iff all the terms in $\mathbf{ran}(\sigma)$ are in (ξ-)normal form. We shall write $\sigma \approx_\xi \theta$ whenever $\sigma(X) \approx_\xi \theta(X)$ for all $X \in V$.

Given a substitution σ and a variable X, the substitution σ_X is defined as:

$$\sigma_X(Y) = \begin{cases} X & \text{if } Y = X; \\ \sigma(Y) & \text{otherwise.} \end{cases}$$

A substitution σ is uniquely extended to a function $\bar{\sigma}$ from τ to τ as follows:

- $\bar{\sigma}(X) = \sigma(X)$, if $X \in V$;
- $\bar{\sigma}(a) = a$, if $a \in \Sigma_0$;
- $\bar{\sigma}(c(t_1, \ldots, t_n)) = c(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n))$, if $c \in \Sigma_n$ $t_i \in \mathcal{T}$; $(1 \leqslant i \leqslant n)$;
- $\bar{\sigma}(\lambda X.t) = \lambda Y.\overline{\sigma_Y}(t)$, where $X \in V$, $Y \in V - \mathbf{var}(\mathbf{ran}(\sigma))$, and $t \in \mathcal{T}$;
- $\bar{\sigma}(t_1(t_2)) = \bar{\sigma}(t_1)(\bar{\sigma}(t_2))$, if $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$.

Since they coincide on V, and for notation convenience, we deliberately confuse a substitution σ and its extension $\bar{\sigma}$. Also, rather than writing $\sigma(t)$, we shall write $t\sigma$. Finally, unless otherwise specified, we shall assume in the sequel that substitutions are in β-normal form, and we shall omit the implicit subscripts "β" in "normal form" and $t\downarrow$, and "$\alpha\eta$" in $\approx$.

Composition is defined as usual up to β-reduction. Given two substitutions $\sigma = \{t_i/X_i\}_{i=1}^n$ and $\theta = \{s_j/Y_j\}_{j=1}^m$, the composition $\sigma\theta$ is the substitution which yields the same result on all terms as first applying σ then applying θ on the result. One computes such a composition as the set:

$$\sigma\theta = (\{t\theta\downarrow/X | t/X \in \sigma\} - \{X/X | X \in \mathbf{dom}(\sigma)\}) \cup (\theta - \{s/Y | Y \in \mathbf{dom}(\sigma)\}).$$

For example, if $\sigma = \{F(X)/Y, V(a)/U\}$ and $\theta = \{a/Y, \lambda X.\lambda Y.Y/F, \lambda X.U/V\}$, then $\sigma\theta = \{\lambda Y.Y/Y, \lambda X.\lambda Y.Y/F, \lambda X.U/V\}$.

Note that this composition modulo β-reduction is partially defined as some reductions may not terminate, in which case we write $\sigma\theta = \perp$. (Take, for example, $\sigma = \{F(F)/Y\}$ and $\theta = \{\lambda X.X(X)/F\}$.) However, provided that all β-reductions terminate, it is clear that if both σ and θ are in normal form then so is $\sigma\theta$.

Composition defines a preorder (*i.e.*, a reflexive and transitive relation) on substitutions. A substitution σ is *more general* than a substitution θ iff there exists a substitution ϱ such that $\theta \approx \sigma\varrho$.

4.2. Unification algorithm

An *equation* is a pair of terms, written $s = t$. A substitution σ is a *solution* (or a *unifier*) of a set of equations $\{s_i = t_i\}_{i=1}^n$ iff $s_i\sigma\downarrow \approx t_i\sigma\downarrow$ for all $i = 1, \ldots, n$. (That is, if all pairs of terms are equal up to $\alpha\beta\eta$-conversion.)

Unless otherwise specified, we shall always assume that all terms in a set of equations are in normal form.

Following [43], we define two transformations on sets of equations—*constructor decomposition* and *variable elimination*. They both preserve solutions of sets of equations. Two sets of equations are *equivalent* iff they both admit all and only the same solutions.

Constructor decomposition

If a set E of equations contains an equation of the form $c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)$, where $c \in \Sigma_n$, $(n \geqslant 0)$, then the set

$$E' = E - \{c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)\} \cup \{s_i = t_i\}_{i=1}^n$$

is equivalent to E[8].

Theorem 2 *Let E be a set of equations containing an equation of the form $c(s_1, \ldots, s_n) = d(t_1, \ldots, t_m)$. If $c \neq d$ or $n \neq m$ then E has no solution. Otherwise, the set E' obtained from E by constructor decomposition using this equation is equivalent to E.*

Proof: If $c \neq d$ or $n \neq m$, then it is clear that no substitution can make the two sides identical. If, on the other hand $c = d$ and $n = m$, then it is also clear that a substitution will be a solution of E iff it is a solution of E'. ■

Variable elimination

If a set E of equations contains an equation of the form $X = t$ where $t \neq X$, then the set

$$E' = (E - \{X = t\})\sigma\downarrow \cup \{X = t\}$$

where $\sigma = \{t/X\}$, is equivalent to E, provided all reductions terminate[9].

Theorem 3 *Let E be a set of equations containing an equation of the form $X = t$ where $t \neq X$. If t is of the form $c(t_1, \ldots, t_n)$, where $c \in \Sigma_n$, and if X occurs free in t then E has no solution. Otherwise, provided that all reductions terminate, the set E' obtained from E by variable elimination using this equation is equivalent to E.*

Proof: If X occurs free in $t = c(t_1, \ldots, t_n)$, then X cannot be made identical to t as

it is its strict subterm. If X does not occur free in t, since by construction $X = t \in E \cap E'$, any solution of E or E' must make in particular X and t identical. Now consider $s = t \in S$ distinct from $X = t$. Let $s' = t'$ be the corresponding equation in E'. Now, if σ makes s and t identical, it must also make s' and t' identical since they may only differ from s and t by the presence of X *versus* t, which are made identical by σ. The same holds conversely, if σ make s' and t' identical. ■

In addition to the two transformations above, the following holds:

Lemma 1 *If a set of equations E contains an equation of the form $\lambda X.s = c(t_1, \ldots, t_n)$ where $c \in \Sigma_n$, $(n \geqslant 0)$, then E has no solution that does not involve at least one η-reduction.*

Proof: Since λ-abstraction and constructor terms are syntactically distinct objects unless related by the rule of η-reduction, this is clearly true. Hence, unless such an η-reduction is performed, no substitution can make them identical. For example the equation $\lambda X.Y(X) = a$ where $a \in \Sigma_0$, has the solution $\{a/Y\}$, but it necessitates η-reduction. ■

Finally, since we need to stop short of synthesizing functional abstractions as done by higher-order unification [31], we shall deliberately ignore solutions which involve solving one or more equations of the form $\lambda X.s = \lambda Y.t$, as well as ignoring η-reduction steps. This is the only source of incompleteness of the equation simplification algorithm presented next.

A Non-deterministic algorithm

A set of equations E is partitioned into two subsets: its *solved* part and its *unsolved* part. The solved part is its maximal subset of equations of the form $X = t$ such that X occurs free nowhere in the full set of equations only as the left hand side of this equation alone. The unsolved part is the complement of the solved part. A set of equations is said to be *fully solved* iff its unsolved part is empty.

The following describes a normalization procedure for a given set E of equations. Repeatedly choose non-deterministically and perform one of the following transformations. If no transformation applies, stop with success.

(a) Select any equation of the form $t = X$ where t is not a variable, and rewrite it as $X = t$.
(b) Select any equation of the form $X = X$ and erase it.
(c) Select any equation of the form $c(s_1, \ldots, s_n) = d(t_1, \ldots, t_m)$ where $c \in \Sigma_n$ and $d \in \Sigma_m$, $(n, m \geqslant 0)$; if $c \neq d$ or $n \neq m$, stop with failure; otherwise, if $n \geqslant 1$ replace it with n equations $s_i = t_i$, $(i = 1, \ldots, n)$.
(d) Select any equation of the form $X = t$ where X is a variable which occurs free somewhere else in the set of equations and such that $t \neq X$. If t is of the form

$c(t_1, \ldots, t_n$, where $c \in \Sigma_n$, and if X occurs free in t, then stop with failure; otherwise, let $\sigma = \{t/X\}$ and replace every other equation $l = r$ by $l\sigma\downarrow = r\sigma\downarrow$;

(e) Select any equation of the form $\lambda X.s = c(t_1, \ldots, t_n)$ where $c \in \Sigma_n$, $(n \geqslant 0)$, and stop with failure;

(f) Select any equation of the form $\lambda X.s = \lambda Y.t$, and stop with failure.

The set of equations which emerges as the outcome of this procedure, if any does, is said to be in *canonical form*. We denote by **can**(E) the set resulting upon termination of the algorithm starting with a set of equations E. If it terminates with failure, we write **can**(E) = $\perp$. Given E in canonical form, its solved part is called its *most general unifier* denoted by **mgu**(E) and its unsolved part is called its *residue* denoted by **res**(E). Elements in **res**(E) are called *residual equations* or *residuations*.

Theorem 4 *Given a set E of equations, provided that all reductions terminate, the foregoing non-deterministic algorithm is such that:* (1) *if E has no unifer then it terminates with failure;* (2) *otherwise it terminates with a canonical set of equations* **can**(E) *such that* **mgu(can**(E)) *is the most general solution of E which does not require solving any equation of the form* $\lambda X.s = \lambda Y.t$ *or performing some* η*-reduction. Moreover, no variable in* **dom(mgu(can**(E))) *occurs free in* **res**(E).

Proof: Steps (a) and (b) trivially preserve all solutions without failure. Steps (c) and (d) are respectively constructor decomposition and variable elimination which preserve all solutions or fail as no unifiers exists, by Theorems 2 and 3. Step (e) fails in the only other case where no unifier exists for E, by Lemma 1. Step (f) discards solutions involving function synthesis, and in particular fails in all cases where such synthesis would fail. Hence, in all cases where unifiers do not exist for E, provided all reductions terminate, the procedure halts with failure.

To establish that, otherwise, the procedure terminates, we make two observations. First, steps (a), (b), and (d) cannot be repeated more times than there are free variables in E. Moreover the equation involved in each of these steps is either eliminated or made ineligible for any other step (a), (b), or (c). Second, constructor decomposition replaces one equation by several, each of strictly shallower depth[10]. Thus, a simple multiset ordering argument [20] is enough to conclude that since depth of terms is well-founded, so is the constructor decomposition procedure. Therefore, provided that all reductions terminate, the algorithm always terminates.

It is clear, by step (f), that **mgu(can**(E)) never requires solving any equation of the form $\lambda X.s = \lambda Y.t$. That it is a most general such unifier of E is also clear since, besides step (f), transformations in all steps preserves all and only solutions to E and thus all ensuing variable assignments are *necessary conditions* for all solutions of E[11]. Finally, by variable elimination in step (d), no variable in **dom(mgu(can**(E))) may occur free in **res**(E). ■

Example

Consider the set of equations

$$\{f(X(Y)(c(a, V)), h(X, a), \lambda U.U) = f(c(Z, b), h(\lambda F.\lambda X.F(F(X)), a), Y)\}$$

A sequence of transformations from the unification algorithm applied to this set is:

(c) $\{X(Y)(c(a, V)) = c(Z, b), h(X, a) = h(\lambda F.\lambda X.F(F(X)), a), \lambda U.U = Y\}$

(a) $\{X(Y)(c(a, V)) = c(Z, b), h(X, a) = h(\lambda F.\lambda X.F(F(X)), a), Y = \lambda U.U\}$

(d) $\{X(\lambda U.U)(c(a, V)) = c(Z, b), h(X, a) = h(\lambda F.\lambda X.F(F(X)), a), Y = \lambda U.U\}$

(c) $\{X(\lambda U.U)(c(a, V)) = c(Z, b), X = \lambda F.\lambda X.F(F(X)), a = a, Y = \lambda U.U\}$

(c) $\{X(\lambda U.U)(c(a, V)) = c(Z, b), X = \lambda F.\lambda X.F(F(X)), Y = \lambda U.U\}$

(d) $\{c(a, V) = c(Z, b), X = \lambda F.\lambda X.F(F(X)), Y = \lambda U.U\}$

(d) $\{a = Z, V = b, X = \lambda F.\lambda X.F(F(X)), Y = \lambda U.U\}$

(a) $\{Z = a, V = b, X = \lambda F.\lambda X.F(F(X)), Y = \lambda U.U\}$

The equation solving procedure described in this section is the unification algorithm on which Le Fun's operational semantics rests. Short of synthesizing functions or recognizing extensionally equivalent functions—neither being really needed in a first-order logic language—Le Fun combines both λ-calculus and predicate logic convenience.

5. Le Fun operational semantics

5.1. Le Fun syntax

We present here a minimal syntax for Le Fun. The idea is not to give an exhaustive description of a "real-life" syntax with all conveniences and sugaring to accommodate aesthetics, but rather to define just enough to focus the reader's attention on the specific originality of Le Fun's syntax—namely, a generalization of applicative expressions and first-order terms. Thus, the reader is assumed to be familiar with Prolog's syntax as well as with basic sugaring of the λ-calculus. Therefore, many unspecified details (*e.g.*, pattern-directed conditionals for functions, handling of functional recursion, *etc.*) are left to the reader's taste after a reading of [40, 5, 53].

Le Fun's terms are a combination of conventional first-order terms and applicative

expressions. More precisely, a Le Fun term is one of the following:

1. *Variables*—represented as capitalized identifiers;
2. *Identifiers*—represented starting with a lower case letter;
3. *Abstractions*—of the form $\lambda X_1 \ldots X_n.e$, where X_i, $i = 1, \ldots, n$ are variables, and e is a Le Fun term;
4. *Applications*—of the form $e(e_1, \ldots, e_n)$, where e is a Le Fun term, and the e_i's are Le Fun terms.

All classical conventions related to left-associativity, infix notation, and currying of applications are assumed. Those special applications of the form $c(e_1, \ldots, e_n)$, where c is an identifier known to be a *constructor* symbol, and the e_i's are Le Fun terms are called *constructions*.

A Le Fun program consists of a sequence of *equations* and *clauses*. An equation is of the form $f = e$ where f is an identifier called an *interpreted symbol*, and e is a Le Fun term. In the case where e is an abstraction of the form $\lambda X_1 \ldots . \lambda X_n.e'$, we may also write $f(X_1, \ldots, X_n) = e'$. A clause is defined exactly as in Prolog, with the difference that Le Fun terms are expected where first-order terms are in Prolog—*i.e.*, as predicate arguments. Such literals which constitute Le Fun clauses will be called Le Fun literals.

The lexical distinction between constructor and interpreted symbols is simply that a constructor is any identifier which does not appear in a left-hand side of an equation. For those, fixed arity is assumed. Hence, any construction with root constructor of arity n must have exactly n arguments. If it has more, the term is ill-formed. If it has less, then the term is not a construction, but an abstraction. Indeed, if c is an n-ary constructor, the term $c(e_1, \ldots, e_k)$ for $k < n$ is in reality the term $\lambda X_1 \ldots . X_{n-k}.c(e_1, \ldots, e_k, X_1, \ldots, X_{n-k})$, where the X_j's do not occur free in any of the e_i's.

Note that in a clause, no confusion should arise between λ-bound and logical variables. The latter are those variables which are not in any λ-scope—occurring free in the clause. Thus, only logical variables are instantiatable by unification[12].

Given a Le Fun program, a *query* is a sequence of Le Fun literals. If only a function evaluation is desired, a query of the form $X = f(e_1, \ldots, e_n)$ will provide the value of evaluating the given functional expression as X's binding. For example, consider the Le Fun program:

```
map(F,L) = if(L = [],
              [],
              [F(head(L))|map(F, tail(L))]).
```

Thus, evaluating a *map* expression in Le Fun is done as:

```
?- X = map(+(1), [0,1,2]).

X = [1,2,3]
```

It is important to understand how function definitions are transformed and stored, and how they are actually used by Le Fun operations. Essentially, one may think of compilation taking place at read time and installing all function definitions in some (interpreted) function symbol store of definitions. These definitions are obtained from the bodies of the functions, perhaps compiled into an appropriate functional machine language (SECD [40], CAM [13], *etc.*). All functional expressions intervening in some other definition bodies, or for that matter some Le Fun clauses or goals, are also compiled as in-line calls (such are clever enough to handle recursion). The functional abstract machine is used in the operational semantics that follows only hidden in the unification process. In that way are the functional definitions used by Le Fun.

5.2. Le Fun operations

Function definitions being transformed away into λ-calculus forms, the operational semantics of Le Fun becomes quite simply identical to Prolog's where unification is replaced by the algorithm of Section 4.2. We formalize this as a state transformation process.

A state of Le Fun computation is either $\bigstar$ (called the *failure* state) or a quadruple $\langle G|E|P|S\rangle$ where G is a sequence of literals called the (goal) *resolvent*, E is a set of equations in canonical form, P is a sequence of Le Fun clauses called the *program*, and S is a state. The computation rule of Le Fun thus is a state transformation relation. We shall write $S_1 \rightsquigarrow S_2$ the transformation from state S_1 to state S_2. Given a clause $H{:-}B$, the notation $(H{:-}B)^\sharp$ denotes that same clause with all its variables consistently renamed with fresh variables. Starting with a state of the form $\langle G|\emptyset|P_0|\bigstar\rangle$, Le Fun proceeds from state to state by repeatedly applying the following computation rules:

1. $\langle G|E|\emptyset|S\rangle \rightsquigarrow S$, if $G \neq \emptyset$;
2. $\langle \emptyset|E|P|S\rangle \rightsquigarrow S$, if $\mathbf{res}(E) \neq \emptyset$;
3. $\langle L, G|E|H{:-}B, P|S\rangle$

$$\rightsquigarrow \begin{cases} \langle L, G|E|P|S\rangle, \\ \text{if } \mathbf{can}(E \cup \{L = H^\sharp\}) = \bot; \\ \langle B^\sharp, G|\mathbf{can}(E \cup \{L = H^\sharp\})|P_0|\langle L, G|E|P|S\rangle\rangle, \\ \text{otherwise;} \end{cases}$$

where $H^\sharp{:-}B^\sharp = (H{:-}B)^\sharp$.

The first transitions are called *backtrack* steps, and the third one is called a *resolution* step. Recall that $\mathbf{can}(E)$ is the canonical form of E computed by the algorithm described in Section 4.2.

This process may either diverge or terminate with a state of one of the following forms:

1. ★, the failure state; or
2. $\langle \emptyset | E | P | S \rangle$ where $\mathbf{res}(E) = \emptyset$. That is, E is fully solved. This is a (full) success state. The solution is $\mathbf{mgu}(E)$. At this point, the same alternative offered by Prolog to try to find other solution may be induced with a backtrack step.

At the state $\langle \emptyset | E | P | S \rangle$ when $\mathbf{res}(E) \neq \emptyset$, one may also choose to stop and display the partial solution $\mathbf{mgu}(E)$ and the residual constraints $\mathbf{res}(E)$, or feed the latter to a special-purpose constraint solver.

6. Implementation of Le Fun

6.1. General principle

The general idea is that residuations can happen at different levels, and that timely and efficient resolution of such residuations can be accomplished through a careful run-time accruement of backchaining information built into a generalized resolution/unification algorithm. Hence, using such run-time information, resolving a residuation should happen automatically when enough information is available for such a resolution to be meaningful (*e.g.*, a residuated functional expression evaluation should be resolved as soon as all the free variables in that expression are ground). One undesired alternative, for obvious reasons, is having to accumulate all residuations in a central repository and check them there periodically for progress potential. The difference between these two alternatives is reminiscent of the difference between *interrupt servicing* and *polling* when a system is dealing with an external signal. The following is a description of the supported residuations and the backchaining information that is deemed necessary for their economical implementation.

At the resolvent level, and as part of a regular goal resolution, a unification can become residuated if a unificand is a function application not ready for evaluation. Therefore, internal representation of function applications must remember the unifications pending on them. Also at the resolvent level, the resolution of ground-decidable predicates can be residuated if their operands are either function applications not ready for evaluation, or uninstantiated variables. Therefore, both function applications and uninstantiated variables should have the capability of remembering the residuated ground-decidable predicates pending on them.

Function applications suspend if free variables therein are still uninstantiated. Therefore, uninstantiated variables should have the capability of remembering the residuated functional evaluations pending on them. We note that, given a function application, partial progress may be possible in reducing such expressions even if all free variables in the expression are not ground. For example, partial computation may allow earlier failures in some computations such as the E-residuation:

$$append([0], X) = append([1], Y)$$

However, the computational overhead needed to support each eager evaluation with the potential of backtracking in considerably more severe, since in general, trailing of all partial evaluations must be kept.

The above points lead us to the following observations:

- Computation fragments that may need to be delayed and remembered (residuated) are (1) functional applications (S-residuations), (2) ground-decidable predicates (I-residuations), and (3) unification operations (E-residuations.)
- Objects that may need to remember residuated computations are: (1) functional applications, and (2) uninstantiated variables.
- The backchaining information is always recorded at unification time, or at the time certain built in predicates are invoked; this is when it is realized whether residuation will be necessary. The unification algorithm will detail the issues related to the nature and placement of that information.
- The backchaining information will be extracted and used at unification time. Failure of released residuated computations simply calls the regular backtracking algorithm, *modulo* a more sophisticated trailing of variable instantiations.

The next section gives some details about an internal representation of Le Fun syntactic objects which we shall use in describing Le Fun's operational semantics.

6.2. *Internal representation*

Objects can be stored in one of two ways: simple objects are stored directly in data cells (boxed objects) with both their value and their tag occupying the same cell. Complex objects, on the other hand, use a two level storage mechanism where the tag and a reference to the actual complex object occupying one data cell, and the complex object itself occupying as much space as needed allocated out of a heap-like storage area.

Simple objects and partial specification of complex objects are shown in the table in Figure 1.

Syntactic Object	*Value Field*	*Tag Field*
Variable Reference	Reference to another variable	Var Ref
Uninstantiated Variable	(Ignored)	Uninst'd
Atomic Object	Atomic value	Atomic
Construction	Reference to the Construction	Constr
Functional Application	Reference to the application	Appl
Functional Abstraction	Reference to the abstraction	Abst
Residuation Variable	Reference to the Var with Resids	Var/Resids

Fig. 1. Internal representation of syntactic objects.

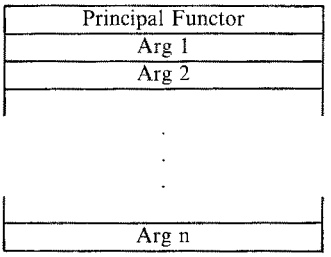

Fig. 2. Construction representation.

Application Expression
Number of Uninst'd Vars
List of Uninst'd Vars
Reduced Value
List of Residuations

Fig. 3. Functional application representation.

Abstraction Expression
Number of Uninst'd Vars
List of Uninst'd Vars

Fig. 4. Functional abstraction representation.

Value (When inst'd)
List of Residuations

Fig. 5. Residuation variable representation.

The complex objects themselves, occupying space allocated out of the heap-like storage, are represented as illustrated in Figure 2 (construction), Figure 3 (functional application), Figure 4 (functional abstraction) and Figure 5 (residuation variable.)

6.3. Dereferencing Le Fun objects

Le Fun unification algorithm will have to recognize, as usual, the basic three data types—uninstantiated variables, atomic objects and constructions. It must also handle functional applications, abstractions, and variables with residuations. Tagging will identify these different structures. Thus, we will assume that every data object in our system consists of two fields—a tag field, and a data field. The unification algorithm can then be visualized as a matrix whose rows and columns are the different types (tag values) of the two unificands. Therefore, the row-column intersections correspond to the unification case particular to the types of the unificands.

We shall use a seventh data type—variable references or pointers. They materialize through unifications between uninstantiated variables. They are made transparent to the unification matrix through the systematic dereferencing of unificands before the actual unification operation. Therefore, such reference chains are left transparent and do not figure explicitly in the matrix. If this produces:

- *an uninstantiated variable*—it is returned.
- *an atomic object*—it is returned.
- *a construction*—it is returned.
- *a functional application*—a check is performed to see if the application is ready for evaluation. If so, dereferencing produces the result; otherwise, the delayed application itself.
- *a functional abstraction*—it is ascertained that the count of uninstantiated variables in the expression was initiated, and the abstraction itself is returned. When the counting of uninstantiated variables starts, incrementing is performed automatically. This is because all the uninstantiated free variables in the expression mutate into uninstantiated variables with residuations (pointing to their common parent expression). These variables are treated specially by the unification algorithm when they become unificands themselves, including maintaining a consistent count of uninstantiated variables in their common parent expression.
- *an RV*—if the variable is unbound, it is returned; otherwise, dereferencing is done recursively on the binding.

Unify Row-Col	Uninst'd Variable	Atomic Object	Const'n Object	Funct'l Appl	Funct'l Abst	Resid'd Variable
Uninst'd Variable	Case 6.4.2					
Atomic Object	Case 6.4.3	Case 6.4.4				
Const'n Object	Case 6.4.3	Case 6.4.1	Case 6.4.5			
Funct'l Appl	Case 6.4.3	Case 6.4.6	Case 6.4.6	Case 6.4.7		
Funct'l Abst	Case 6.4.3	Case 6.4.1	Case 6.4.1	Case 6.4.6	Case 6.4.1	
Resid'd Variable	Case 6.4.3	Case 6.4.8	Case 6.4.8	Case 6.4.9	Case 6.4.9	Case 6.4.10

Fig. 6. Unification matrix for Le Fun objects.

6.4. Implementing Le Fun unification

Le Fun's unification algorithm is better expressed in the form of a matrix (see Figure 6) where the rows and columns correspond to the different types of Le Fun terms, and the row-column intersections specify the specialized treatment of the corresponding unification case. By symmetry, only half of the matrix is presented.

The treatment of the different unification cases is summarized in the following subsections.

6.4.1. Failure. The treatment of failures here is very similar to the treatment of failures in conventional logic programming systems. This means backtracking to the most recent choice point where alternatives still exist, and proceeding from there after the system is restored to its previous state. The difference comes from the fact that in conventional systems the only thing that can happen to an uninstantiated variable is becoming instantiated; whereas here, such a variable may mutate into an RV (see Cases 6.4.6 and 6.4.7.) Then, both types of changes must be remembered for potential failure, and thus, backtracking. In the latter case, restoring uninstantiated variables to their original state is assumed for both variables that became instantiated as well as variables that mutated to RVs.

A noteworthy point is that we could relax some failures of Le Fun unification should we decide to use narrowing to set a (perhaps higher-order) deductive mode. That is, we may choose to use first-order equation solving or higher-order unification to deal with some of the cases involving functional abstractions or applications as a unificand in the matrix above. All variations of this principle are naturally to be considered—*i.e.*, declaring some functors to be narrowable—even if only for intelligent debugging and trouble-shooting.

6.4.2. Variable versus variable. This is the simplest kind of residuation, and boils down to dereferencing unificands before the unification operation is attempted—*vz.*, making one of the variables point to the other, and tag it as a *variable reference.* This case, of course, will always succeed.

6.4.3. Variable versus non-variable. This is the same as the above case, except that it can be optimized by overwriting the uninstantiated unificand with the non-variable one itself rather than with a pointer to it. Here, as in 6.4.2, unification always succeeds —subject to *occur-check.*

6.4.4. Atom versus atom. This is simply an equality check between the two atomic unificands.

6.4.5. Construction versus construction. This case consists of a simple equality check between the functors (including their arity). This is followed, if successful, by the recursive unification of the unificands' respective arguments.

6.4.6. Application versus atom, construction, or abstraction. This case and Case 6.4.7 are that of the unification between delayed functional applications and other objects. This is the simpler of two possible cases, where the other unificand is not another application. It creates two kinds of residuations:

1. One, recorded in the application itself, remembering this present unification that cannot be completed pending the evaluation of the application unificand; and,

2. another one, obtained by mutation of every uninstantiated variable in the application into an RV. Note that "uninstantiated variable," in this context, includes RVs; in which case the new residuation is simply added to an already existing chain of older ones.

This unification case succeeds provisionally, and never leads to immediate failure. Latent failures will be treated when they materialize.

6.4.7. Application versus application. This case is identical to the previous one except that the same treatment as described then is applied to both (functional application) unificands.

6.4.8. RV versus atom or construction. When an RV is instantiated, a chain of events is triggered based on the type of the other unificand. The present case is the simpler of such cases, and deals with a second unificand which is either an atomic object or a construction. Then, the residuations pending on this variable are either applications or ground-decidable predicates. Reference counting reduces the number of RV's by one. Zero such RV's trigger evaluation, or the obvious decision in the case of a ground-decidable literal.

6.4.9. RV versus application or abstraction. Instead of simply substrating one as before from the RV count, we add $n - 1$ to that number where n is the number associated with the other unificand.

6.4.10. RV versus RV. The value field of one of the two RVs is used to point to the other, and its chain of residuations is appended to the other's chain.

Overhead in recording residuations and resolving them is incurred only in the most general case. In the simple case where nothing is residuated, unification is obviously as efficient as the conventional one.

7. Relation to other works

Our computation model has two characteristics: first, the integration of (Horn clause resolution based) logic programming and (λ-calculus based) functional programming, and second the introduction of a powerful implicit asynchronous control strategy into a practical programming system. In addition, we found residuation to be a good vehicle for implementing native operating systems capabilities (coroutines, I/O drivers, schedulers, and general interrupt handling primitives are examples of such capabilities).

Relatively recent incarnations of Prolog, *e.g.*, MU-Prolog [50] and Prolog-II [11], handle such problems by giving explicit flow information hints to the interpreter or compiler. In these, a user can delay the evaluation of specific ground-decidable predicates (*e.g.*, inequality: $\cong$ in MU-Prolog and `dif/2` in Prolog-II) until the

arguments become ground. Thus, as a first approximation, residuation may be simply described as a generalization on these and the similar, albeit more restricted because explicit, concepts of *read-only variables* of Concurrent Prolog [59], or Parlog's *mode declarations* [9] and Prolog-II's *freeze*—all using explicit annotations and/or meta-predicates. However, the extent to which this generalization has been carried out, including doing away with user-supplied control annotation delaying goal resolution, gives Le Fun quite a different taste.

Another strongly related paradigm is the notion of I-structures in ID [51] the MIT dataflow language. I-Structures are write-once random-access data cells. After an I-structure is allocated, one may manipulate it (or rather refer to it) before information is stored in the cell. If the cell's content is needed prior to its materialization, the requesting computation is suspended. It is resumed at the time the cell becomes filled. No overwriting—even consistent—is allowed into I-structures. Their purpose, of course, is only incidentally connected to ours, and pertains to augmenting a (dataflow) functional language with arrays.

There are other language proposals based on extensions of the λ-calculus which attempts to capture operational features of Prolog (essentially, unification and backtracking). QUTE [58] and FRESH [60] are two such instances. These two languages are similar to one another in that they are both built upon a pattern-oriented λ-calculus where matching has been replaced by unification. They are similar to ours in that neither uses higher-order unification. However, they differ substantially from Le Fun and from the languages reviewed in the survey of Section 2.2 in that they do not literally integrate *predicate logic* programming with functional programming. More precisely, programs in QUTE or FRESH do not have a logical reading as Prolog programs do. They consist of functional expressions which may alter a global environment. This is why we did not review them as members of the four proposed categories of integration since they are not strictly speaking integrating logic and functional programming. Nevertheless, they stand out among current research as quite original.

Finally, the operational scheme of Le Fun may be seen through the general approach of Constraint Logic Programming (CLP) due to Jaffar and Lassez [34]. They develop a semantic, algebraic, and operational scheme extending logic programming as seen in Prolog, where unification on first-order terms is generalized to constraint solving in arbitrary but solvable domains such as of linear equations, inequations with real coefficients, boolean formulas, infinite trees, *etc.* Conventional first-order term unification thus turns out to be a simple instance of solving syntactic equational constraints. The general CLP operational scheme proceeds by transforming constraint sets into solved forms (*e.g.*, unification) or canonical forms which are solvable by special purpose algorithms (*e.g.*, Gaussian elimination, Simplex method, *etc.*). It is clear that Le Fun's operational semantics falls into this scheme in that Le Fun term unification is a particular constraint solving mechanism. Thus, we foresee that it should inherit the abstract model-theoretic properties developed by Jaffar and Lassez. That would constitute a formal semantics for Le Fun and such is a topic for further work.

8. Conclusion

Long live freedom!
Anonymous, *Unpublished, undated.*

In this article we reviewed a number of important proposals for the integration of logic and functional programming. We examined the motivation and reason for desiring such an integration, staying away from the conventional prejudices for or against either style of programming. We attempted to categorize roughly the main approaches relatively to our stated *desideratum*, and highlighted some particular points of interest. We introduced a new paradigm for integrating resolution-based and λ-calculus based programming which consists of a delaying mechanism built into the unification process to account for β-reduction. We coined the work *"residuation"* to describe this method as it can be formalized as an equation transforming process allowing equational residues to wait for materialization of information to be solved further. The usefulness of residuation for handling ground-decidable predicates was a straightforward generalization. Much more work remains to be done as far as semantics and extension to a real programming language are concerned. Nevertheless, we have described an operational semantics and implementation for a prototype programming language called Le Fun to validate our design concepts.

Appendix: Le Fun as a constraint language

Residuation and the ensuing integration of logic and functional programming can be characterized in different ways.

One of these views Le Fun as adequately fit for constraint problems. Indeed, those which can be solved by pre-posting constraints whose evaluation, possibly involving functional reductions, is delayed until more information subject of the constraints comes. A large class of symbolic processing problems can be thought of as consisting of a search through feasible states. Feasibility of such states is done as constraint checking. Failing the constraints could trigger a return to the search phase followed again by another constraint checking phase, and so on. Such an approach could be shown to be in general computationally wasteful. The wasteful computation usually results from working on generating the rest of the states when one is already doomed to cause a future constraint check failure.

One immediate improvement is the interweaving of the generation of state instantiations and the checking of constraints. We claim, however, that a more natural and efficient way for dealing with this problem is to be able to pre-post the constraints, even though they are not yet decidable, and resolve them as instantiations are incrementally generated, thus relieving the programmer from the explicit interleaving of the instantiation and the constraint checking phases.

A.1. Cryptarithmetic

Our first example is this well-known cryptarithmetic puzzle where the solution consists of (decimal digits) assignments for the letters S, E, N, D, M, O, R, Y that would make the following addition operation arithmetically correct:

$$\begin{array}{r} \quad SEND \\ + \quad MORE \\ \hline = MONEY \end{array}$$

A (relatively) declarative specification of this problem is to say that the letters in question stand for decimal digits such that such incremental (arithmetic) constaints are obeyed. The constraints here involve relationships between functions involving the letters in question as well as intermediate variables (carry digits $C1$, $C2$, and $C3$.)

One way that can be be written in Prolog:

```
solution(S,E,N,D,M,O,R,Y)
:- % Generating decimal digits:
   decimal_digits([S, E, N, D, M, O, R, Y]),
   % Generating binary digits:
   zero_or_one(C1),
   zero_or_one(C2),
   zero_or_one(C3),
   % The arithmetic constaints:
   C3 + 2 + M = O + 10*M,
   C2 + E + O = N + 10*C3,
   C1 + N + R = E + 10*C2,
        D + E = Y + 10*C1.
```

The problem with this approach is that as candidate assignments are being generated (within `decimal_digits/1`) for the letters, partial results may already be doomed to cause failure but the *search* goes on to instantiate the rest of the candidates. The search space for the solution of our problem is therefore unnecessarily large, and we would like to have natural ways of trimming it with minimal user effort. Why not post the constraints up front, assuming the system allows such literal ordering to produce a successful execution?

The problem of course is that the unification between two functional expressions (*e.g.*, $D + E$ and $Y + 10*C1$ in this example) is only conceivable in conventional Prolog if the two expressions can be reduced to canonical forms so that unification may decide. When delayed computation fragments become executable and result in a failure, chronological backtracking is used to explore available alternatives. The point of failure is, of course, the instantiation that triggered the failed residuated computation. Thus, the new version of the same program as above is:

```
solution(S,E,N,D,M,O,R,Y)
  :- % The arithmetic constraints:
     C3 + S + M = O + 10*M,
     C2 + E + O = N + 10*C3,
     C1 + N + R = E + 10*C2,
          D + E = Y + 10*C1,
     % Generating binary digits:
     zero_or_one(C1),
     zero_or_one(C2),
     zero_or_one(C3),
     % Generating decimal digits:
     decimal_digits([S, E, N, D, M, O, R, Y]).
```

A.2. Architecture constraints

This example exhibits the capability of delaying inequality checks (and ground-decidable predicates in general) until such checks are possible (the necessary variables become instantiated). Hence, the inequality predicate $\cong$ is also subject to residuation when any of its operands still has uninstantiated variables. This is the special case of residuation that corresponds to Prolog-II's `dif/2` and to MU-Prolog's $\cong$. Here again then, the computation model allows a constraint to be pre-posted and makes it possible to resolve such constraint checking asynchronously as it becomes possible to process. The problem in question, taken from [10], can be verbally defined as follows:

Design architectural units given the following design rules:

1. units consist of two rooms;
2. one room, the *front-room*, has an exterior door;
3. rooms have an interior door and a window;
4. rooms are connected by the interior door;
5. walls can have only one opening in them (doors or windows);
6. no windows should be on the north side;
7. windows should not be on opposite sides of the unit.

The problem can be expressed as follows. The solution instantiations are direction specification for the units components:

```
unit(Exterior_Door,
     Room1_Door,
     Room1_Window,
     Room2_Door,
     Room2_Window)
```

```
:- % constraint 5:
   ~=(Exterior_Door, Room1_Door, Room1_Window),
   % constraint 5:
   Room2 Door ~= Room 2 Window,
   % constraint 4:
   opposite(Room1_Door,Room2_Door),
   % constraint 7:
   not_opposite(Room1_Window,Room2_Window),
   % constraint 6:
   Room1_Window ~= north,
   Room2_Window ~= north,
   % Candidate assignments:
   directions([Exterior_Door,
               Room1_Door,
               Room1_Window,
               Room2_Door,
               Room2_Window]).
```

and

```
opposite(east,west).
opposite(west,east).
opposite(north,south).
opposite(south,north).

not-opposite(Dir1,Dir2)
   :- opposite(Dir1,Dir3),
      Dir2 ~= Dir3.
```

With a runtime environment that allows the successful execution of such a literal ordering, the effect of this program statement is to pre-post the design constraints (which will get residuated) and then the search space is truncated every time a partial assignment triggers the failure of a residuated constraint check. The search space truncation corresponds to the useless generation of the assignments complementary to the partial ones causing the constraint violation. Our contention is that the efficiency of the asynchronous computation model is superior to a sequential *generate and test* model.

Acknowledgments

The authors are indebted to Patrick Lincoln for many pertinent conversations and his quick mind in general. Much credit is also deserved by the anonymous referee for her/his sharp observations and for pointing out several technical inconsistencies. Thanks finally to Jan Zubkoff the associate editor of *LISP* for her encouragements.

Notes

1. Which may include taste—*our taste.*
2. The reader solely interested in Le Fun may wish to skip directly to Section 2.3 on a first reading. The survey is intended for the reader who may desire to put our work in context. For additional details, all interested reaer is referred to the literature, which is literally bursting with publications on unifying logic and functional programming. She will find a good sample in [6, 17, 36], as well as in various proceedings of symposia, conferences, and workshops on functional or logic programming.
3. Narrowing [21, 33, 56] is term-rewriting where matching is replaced by unification, thereby giving function application the power to synthesize uninstantiated arguments from patterns in the definitions. Innermost narrowing is to narrowing what applicative (innermost) order of reduction is to normal (outermost) order of reduction. Thus, some notion of function strictness must be used to ensure good behavior. As in the case of reduction, it is a more efficient equation-solving method when used on strict terms, but may diverge on non-strict ones (see [62, 44]).
4. Smolka's logic is restricted to what he calls *canonical* equational systems in which equations are of the form $f(t_1, \ldots, t_n)$ where the t_i's are restricted to be constructor terms; *i.e.*, contain no defined symbols —roots to some left-hand sides—such as f. By virtue of being a root of no right-hand side, a symbol is a 'constructor.'
5. For higher-order varieties, see [24].
6. Unless the set of equations which define them is a complete set of reductions, or can be completed by the Knuth–Bendix method–which may itself diverge if no finite complete set of reductions exists.
7. Types (whether Church's simple types [8] or polymorphic [48]) are orthogonal here. We shall ignore them in this presentation, although they are of importance for compilation and computation.
8. If $n = 0$, the equation is simply deleted.
9. If $E = \{s_i = t_i\}_{i=1}^n$ then $E\sigma\downarrow = \{s_i\sigma\downarrow = t_i\sigma\downarrow\}_{i=1}^n$.
10. The depth of an equation is the greater of the depths of each of its sides, where depth of a term is defined as usual.
11. That is, it is most general up to variable renaming since an equation of the form $X = Y$ with $X \neq Y$ in step (d) chooses arbitrarily to eliminate X rather than Y.
12. For example, following Landin [40] applicative expressions may be compiled into SECD virtual machine code, and thus λ-bound variables are translated into displacements indices corresponding to their binding heights. Other applicative programming techniques, like combinator reduction, would as well eliminate those variables which are λ-bound.

References

1. Aït-Kaci, H. and Nasr, R. *Residuation: A Paradigm for Integrating Logic and Functional Programming.* MCC Technical Report Number AI-359-86, Microelectronics and Computer Technology Corporation, Austin, TX (October 1986).
2. Barbuti, R. *et al.* LEAF: a language which integrates logic, equations, and functions. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations,* Prentice-Hall, Englewood Cliffs, NJ (1986) 201–238.
3. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam (1981).
4. Bosco, P. G. and Giovanetti, E. *A Prolog-Compiled Higher-Order Functional and Logic Programming Language.* Technical Draft, CSELT, Torino, Italy (1986). An abridged version of this paper has appeared in the proceedings of the 3rd IEEE Symposium on Logic Programming, Salt Lake City, UT (September 1986).
5. Burge, W. H. *Recursive Programming Techniques.* Addison-Wesley, Reading, MA (1975).
6. Campbell, J. A. *Implementations of Prolog.* Ellis Horwood, Ltd., Chichester, UK (1984).
7. Chew, L. P. An improved algorithm for computing with equations. In *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science,* Syracuse, NY (1980) 108–117.

8. Church, A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, **5** (1940) 56–68.
9. Clark, K. L. and Gregory, S. *Parlog: A Parallel Logic Programming Language*. Research Report DOC-83/5, Department of Computing, Imperial College, London, UK (May 1983).
10. Coehlo, H., Cotta, J. C., and Pereira, L. M. *How to solve it with Prolog*. Technical Report, Ministério da Habitação e Obras Públicas, Laboratorio de Engenharia Civil, Lisbon, Portugal (1980).
11. Colmerauer, A. *et al. Prolog II: Reference Manual and Theoretical Model*. Technical Report, Groupe d'Intelligence Artificielle, Faculté des Sciences d'Aix-Luminy, Marseille, France (1982).
12. Courcelle. B. Fundamental properties of infinite trees. *Theoretical Computer Science*, **25** (1983) 95–169.
13. Cousineau, G., Curien, P. L. and Mauny, M. The categorical abstract machine. In Jouannaud, J.-P., editor, *Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag, Berlin, W. Germany (September 1985) 50–64.
14. Curien, P. L. *Categorical Combinators, Sequential Algorithms and Functional Programming. Research Notes in Theoretical Computer Science*, Pitman, London, UK (1986).
15. Damm, W. Languages defined by higher program schemes. In *Proceedings of the 4th International Conference on Automata, Languages, and Programming*, LNCS 52, Springer-Verlag, Berlin, W. Germany (1977).
16. Darlington., J. The unification of functional and logic programming. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, Englewood Cliffs, NJ (1986) 37–72.
17. DeGroot, D. and Lindstrom, G., editors. *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, Englewood Cliffs, NJ (1986).
18. Dershowitz, N. *Applications of the Knuth-Bendix Completion Procedure*. Aerospace Report ATR83(8478)-2, Lab. Operations, The Aerospace Corporation, El Segundo, CA (May 1983).
19. Dershowitz, N. Completion and its applications. In Aït-Kaci, H. and Nivat, N., editors, *Resolution of Equations in Algebraic Structures*, Academic-Press, Boston, MA (Forthcoming in 1989).
20. Dershowitz, N. and Manna, Z. Proving termination with multiset ordering. *Communications of the ACM*, **22**, 8 (1979) 465–476.
21. Fay, M. First-order unification in an equational theory. In *Proceedings of the 4th Conference on Automated Deduction*, Austin, TX (1979) 161–167.
22. Felleisen, M. *Transliterating Prolog into Scheme*. Technical Report 182, Computer Science Department, Indiana University, Bloomington, IN (1985).
23. Fribourg, L. Handling function definition through innermost superposition and rewriting. In Jouannaud, J.-P., editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, LNCS 202, Springer-Verlag, Berlin, W. Germany (May 1985) 325–344.
24. Gallier, J. H. *n*-Rational algebras, part I: basic properties and free algebras. *SIAM Journal on Computing*, **13**, 4 (November 1984) 750–775.
25. Gallier, J. H. *n*-Rational algebras, part II: varieties and logic of inequalities. *SIAM Journal on Computing*, **13**, 4 (November 1984) 776–794.
26. Gallier, J. H. *Logic for Computer Science: Foundations of Theorem-Proving*. Harper & Row, New York, NY (1986) chapter 10, §6: Decision Procedures Based on Congruence Closure, 461–474.
27. Goguen, J. and Meseguer, J. *An Initiality Primer*. Technical Draft, Computer Science Laboratory, SRI International, Menlo Prk, CA (March 1983).
28. Goguen, J. and Meseguer, J. Eqlog: equality, types, and generic modules for logic programming. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, Englewood Cliffs, NJ (1986) 295–364.
29. Goldfarb, W. D. The undecidability of the second-order unification problem. *Theoretical Computer Science*, **13** (1981) 225–230.
30. Hsiang, J. and Dershowitz, N. Rewrite methods for clausal and nonclausal theorem proving. In *Proceedings of the 10th International Conference on Automata, Languages and Programming*, LNCS 154, Springer-Verlag, Berlin, W. Germany (1983) 331–346.
31. Huet, G. *Constrained Resolution: A Complete Method for Higher-Order Logic*. PhD thesis, Department of Computing and Information Sciences, Case Western Reserve University (August 1972).
32. Hughes, J. *Graph Reduction with Super-Combinators*. Technical Monograph PRG-28, Programming Research Group, Oxford University, Oxford, UK (1982).

33. Hullot, J.-M. Canonical forms and unification. In *Proceedings of the 5th Conference on Automated Deduction*, LNCS 87, Springer-Verlag, Berlin, W. Germany (1980) 318–334.
34. Jaffar, J. and Lassez, J.-L. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, W. Germany (January 1987).
35. Josephson, A. and Dershowitz, N. An implementation of narrowing: the RITE way. In *Proceedings of the 3rd IEEE Symposium on Logic Programming*, Salt Lake City, UT (September 1986) 187–199.
36. Jouannaud, J.-P., editor. *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*. Volume 202 of *LNCS*, Springer-Verlag, Berlin, W. Germany (May 1985).
37. Jouannaud, J.-P. and Kirchner, H. *Completion of a Set of Rules Modulo a Set of Equations*. Rapport de Recherche 84-R-046, CRIN, Vandœuvre-lès-Nancy, France (1985). A preliminary version of this paper was presented at the 11th ACM Symposium on Principles of Programming Languages, in Salt Lake City, UT (1984).
38. Kirchner, C. Computing unification algorithms. In *Proceedings of IEEE Computer Society Symposium on Logic in Computer Science*, Cambridge, MA (June 1986) 206–217.
39. Klop, J. W. Term rewriting systems. Lecture Notes, Seminar on Reduction Machines, Ustica, Italy (September 1985).
40. Landin, P. J. The mechanical evaluation of expressions. *Computer Journal*, **6**, 4 (1963) 308–320.
41. Lincoln, P. D. *DisCoRd: Distributed Combinator Reduction*. Bachelor Thesis, Department of EECS, Massachussets Institute of Technology, Cambridge, MA (May 1986).
42. Mahr, B. and Makowsky, J. A. Characterizing specification languages which admit initial semantics. *Theoretical Computer Science*, **31** (1984) 49–60.
43. Martelli, A. and Montanari, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**, 2 (April 1982) 258–282.
44. Martelli, A., Rossi, G. F., and C., Moiso. Lazy unification algorithms for canonical rewrite systems. In Aït-Kaci, H. and Nivat, N., editors, *Resolution of Equations in Algebraic Structures*, Academic-Press, Boston, MA (Forthcoming in 1989).
45. Mellish, C. and Hardy, S. Integrating Prolog in the PopLog environment. In Campbell, J. A., editor, *Implementations of Prolog*, Ellis Horwood, Ltd., Chichester, UK (1984) 147–162.
46. Meseguer, J. and Goguen, J. A. Initiality, induction, and computability. In Nivat, M. and Reynolds, J., editors, *Algebraic Methods in Semantics*, Chapter 14, Cambridge University Press, Cambridge, UK (1985) 459–541.
47. Miller, D. A. and Nadathur, G. Higher-order logic programming. In Shapiro, E., editor, *Proceedings of the 3rd International Conference on Logic Programming*, LNCS 225, Springer-Verlag, Berlin, W. Germany (July 1986) 448–462.
48. Milner, R. A theory of type polymorphism in programming. *Journal of Computing Systems and Science*, 17, **3**, (December 1978) 348–375.
49. Nadathur, G. *Higher-Order Logic Programming and Applications*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (December 1986).
50. Naish, L. *MU-Prolog 3.1db Reference Manual*. Computer Science Department, University of Melbourne, Melbourne, Australia (May 1984).
51. Nikhil, R., Pingali, K., and Arvind. *ID Nouveau*. Computational Structures Group Memo 265, Massachussets Institute of Technology, Cambridge, MA (July 1986).
52. Peterson, G. and Stickel, M. Complete sets of reductions for some equational theories. *Journal of the ACM*, **28** (1983) 233–264.
53. Peyton Jones, S. L. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ (1987).
54. Ramakrishnan, R. and Silberschatz, A. Annotations for distributed programming in logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, St-Petersburg Beach, FL (January 1986) 255–262.
55. Reddy, U. On the relationship between logic and functional programming. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, Englewood Cliffs, NJ (1986) 3–36.

56. Réty, P. *et al.* Narrower: a new algorithm for unification and its application to logic programming. In Jouannaud, J.-P., editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, Springer-Verlag, Berlin, W. Germany (1985) 141–157.
57. Robinson, J. A. and Greene, K. J. *New Generation Knowledge Processing: Final Report on the SUPER System.* CASE Center Technical Report No. 8707, Syracuse University, Syracuse, NY (May 1987).
58. Sato, M. and Sakurai, T. QUTE: a functional language based on unification. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, Englewood Cliffs, NJ (1986) 131–155.
59. Shapiro, E. *A Subset of Concurrent Prolog and its Interpreter.* Technical report TR-003, Institute for 5th Generation Computing, Tokyo, Japan (January 1983).
60. Smolka, G. FRESH: a higher-order language with unification and multiple results. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, Englewood Cliffs, NJ (1986) 469–524.
61. Smolka, G. Some thoughts on logic programming. Lecture Notes, Microelectronics and Computer Technology Corporation, Austin, TX (July 23 1986).
62. Smolka, G. *TEL Version 0.9: Report and User Manual.* SEKI Report ST-87-11, Universität Kaiserslautern, Kaiserslautern, W. Germany (February 1988).
63. Turner, D. A. A new implementation technique for applicative languages. *Software—Practice and Experience*, **9** (1979) 31–49.
64. Turner, D. A. Recursion equations as a programming language. In Darlington, J., Henderson, P., and Turner, D. A., editors, *Functional Programming and its Applications: An Advanced Course*, Cambridge University Press, Cambridge, UK (1982) 1–29.
65. Warren, D. H. D. Higher-order extensions of Prolog—are they needed? In Michie, D., editor, *Machine Intelligence 10*, Edinburgh University Press, Edinburgh, UK (1982) 441–454.
66. Yelick, K. Combining unification algorithms for confined equational theories. In Jouannaud, J.-P., editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, Springer-Verlag, Berlin, W. Germany (1985) 365–380.
67. You, J. H. and Subrahmanyam, P. A. Equational logic programming: an extension to equational programming. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, St-Petersburg Beach, FL (January 1986) 209–218.