Unification on the Run

Thomas Prokosch and François Bry

Institute for Informatics, Ludwig-Maximilian University of Munich, Germany
prokosch@pms.ifi.lmu.de bry@lmu.de

1 Introduction

Since Robinson introduced unification [6], many variations of Robinson's unification algorithm [6] have been proposed [5, 4, 1, 2, 3]. Indeed, "[t]he unification algorithm as originally proposed can be extremely inefficient" [4, page 259]. Improving over Robinson's original unification algorithm has been attempted in three manners:

- by changing when the occurs check is performed (as done for example in [4]),
- by sharing instead of copying subexpressions to which variables are bound (as first suggested by [5, 1]),
- by simplifying the already computed substitution or the expressions still to unify (as done with the rules "variable elimination," "reduction" and "compactification" of [4]).

An issue which has received little attention is whether a potential improvement of a unification algorithm is realizable in run-time systems. Suggestions of the afore-mentioned third kind seem hardly realizable at reasonable costs in a runtime system. The article [3] stresses that many suggestions for improving Robinson's unification algorithm are not successful in practice. That article reports on an empirical evaluation of the performance of the unification algorithms by Robinson [6], Martelli-Montanari [4], Escalada-Ghallab [2], and a formerly unpublished improvement of Robinson's algorithm showing that, unexpectedly, Robinson's unification algorithm is the most efficient!

This article reports on a refinement of Robinson's original unification algorithm based on

- an in-memory representation of expressions, an issue not considered by Robinson,
- single left-to-right runs through, or traversals, of the expressions tested for unifiability,
- keeping track of the matching or unification mode of the sub-expressions so far run through, an approach so far not considered,
- and exploiting the afore-mentioned in-memory representation for detecting when occurs checks are unnecessary.

2 Preliminaries

Finitely many non-variable symbols and infinitely many variables are considered. In the following, the lower case letters $a, b, c, \dots, z$ with the exception of v denote the non-variable symbols, $v_0, v_1, v_2, \dots$ (with subscripts) denote the variables. v^i (with a superscript) denotes an arbitrary variable.

An *expression* is either a first-order term or a first-order atomic formula. Expressions are defined from constructors as follows. A *constructor* is a pair s/a with s a non-variable symbol and a one of finitely many arities associated with the symbol s. There are finitely many constructors. An *expression* is either a *variable* or a *non-variable expression.* A non-variable expression is either a constructor s of arity 0, $s/0$, or it has the form $s(e_1, \dots, e_n)$ where s/n is a constructor of arity $n \geq 1$ and $e_1, \dots, e_n$ are expressions. $e_1, \dots,$ and e_n are the *direct subexpressions* of $s(e_1, \dots, e_n)$. Two expressions are *variable-disjoint* if none of the variables occurring in the one expression occurs in the other.

An expression $s(e_1, \dots, e_n)$ is in *standard notation.* It can also be written without parentheses in *prefix notation* (or *Polish* or *Łukasiewicz notation*) as $s/n\ e_1/a_1\ \dots\ e_n/a_n$ where a_i is the arity of expression e_i $(1 \leq i \leq n)$. While the standard notation is easier to read the (parenthesis-free) prefix form is necessary for linear (or parenthesis-free) processor languages.

3 In-memory representations and dereferencing

The representation of an expression in the memory of a run-time system is based on the expression's prefix notation. Assuming that a constructor and a variable are stored in 4 bytes and storage begins at address 0, the representation of $f(a, v_1, b, v_1)$ is:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19
$f/4$	$a/0$	nil	$b/0$	8

The leftmost, or first, occurrence of the variable v_1 is represented by the value nil which indicates that the variable is unbound. The second occurrence of the variable v_1 is represented by an offset: The address of this second occurrence's representation, 16, minus the offset, 8, is the address of the representation of the variable's first occurrence, 8. Occurrences of (the representation of) a variable like the second occurrence of v_1 in $f(a, v_1, b, v_1)$ and the cell representing such variables like the cell at address 16 in the above representation of $f(a, v_1, b, v_1)$ are called a *locally bound variables* or *offset variables.*

Two properties of an expression representation are worth stressing:

1. Variables' names are irrelevant to expression representations, that is, variant expressions have the same representation except for the memory addresses.
2. Two distinct expression representations do not share variables.

Representation of substitutions An elementary substitution $\{v^i \mapsto e\}$ can be seen as a pair (address of v^i, address of the representation of e). If the representation of $p(a, v_1, v_1)$ is stored at address 0 and the representation of $q(b, v_3)$ at address 23, the substitution application $p(a, v_1, v_1)\{v_1 \mapsto q(b, v_3)\}$ is represented before substitution application as:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19 20 21 22	23 24 25 26	27 28 29 30	31 32 33 34
$p/3$	$a/0$	nil	4		q/2	b/0	nil

and after the application of $p(a, v_1, v_1)\{v_1 \mapsto q(b, v_3)\}$ as:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19 20 21 22	23 24 25 26	27 28 29 30	31 32 33 34
$p/3$	$a/0$	23	4		q/2	b/0	nil

Observe that the cell representing the second occurrence of the variable v_1 (cell at 12) keeps its offset (4) unchanged. Thus, binding a variable v which occurs in an expression e to an expression e' consists in storing at the leftmost occurrence of v in the representation of e the address of the representation of e', leaving unchanged further occurrences of v in the representation of e. This approach to binding variables makes the representation of a substitution application unique.

Dereferencing Consider the following two representations of $f(a, v_1, v_2, v_1, g(v_1))$:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27
$f/5$	$a/0$	nil	nil	8	g/1	16

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27 28 29 30 31 32 33 34 35	36 37 38 39	40 41 42 43
$f/5$	$a/0$	nil	nil	8	36		g/1	8

The first representations of $f(a, v_1, v_2, v_1, g(v_1))$ is dereferenced because, except for the representations of the second and third occurrences of the variable v_1, the variables' value are nil. The second and third occurrences of v_1 cannot be dereferenced like a pointer because this would result in the following representation of $f(a, v_1, v_2, v_3, g(v_4))$, not of $f(a, v_1, v_2, v_1, g(v_1))$:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27 28 29 30 31 32 33 34 35	36 37 38 39	40 41 42 43
$f/5$	$a/0$	nil	nil	nil	36		g/1	nil

A dereferenced representation of an expression e is generated from any representation R_e of e as follows: While traversing R_e from left to right, if the cell reached contains a constructor or nil, or the offset of a locally bound variable, then copy the cell's content to a new cell. Otherwise (the token reached is a non-locally bound variable v storing the address E of an expression representation), recursively dereference the expression representation at address E.

In dereferencing, care must be given not to trespass expression representations' ends in recursive calls. This is cared for by using as follows the constructors' arities during a left-to-right traversal of an expression representation E: Let R denote the number of remaining (sub-)expression representations; set $R := 1$ before traversing E, at each constructor s/n perform the update $R := R - 1 + n$ (-1 for the (sub-)expression beginning at that constructor, and $+n$ for the n subexpression representations now to be traversed), and at each variable perform the update $R := R - 1$. The expression representation's end is reached when $R = 0$.

4 A matching-unification algorithm

A call to `unif(e1, e2)` performs a left-to-right run through, or traversal, of the representations of expressions e_1 and e_2 stored at the addresses `e1` and `e2` respectively. The algorithm makes uses of the variables

- `A`: One of `VR` (variant), `SI` (strict instance), `SG` (strict generalisation), `OU` (only unifiable, i.e. unifiable but none of `VR`, `SI`, `SG`), or `NU` (not unifiable). Initialisation: `A := VR`
- `R1` and `R2`: The end of the expression representation at address `e1` (`e2`, respectively) is reached when `R1 = 0` (`R2 = 0`, respectively). Initialisation: `R1 := arity(e1); R2 := arity(e2)`
- `S1` and `S2`: Substitutions for variables in the expression representations at addresses `e1` and `e2` respectively. Initialisation: `S1 := []; S2 := []`

`S+R` denotes the list obtained by appending `R` to the list `S`. `a += b` is shorthand for `a := a+b`. The algorithm make uses of the functions:

- `type(e)`: Type of the value stored in the cell with address `e`: `cons` if that value is a constructor `s/n`, `novar` if it is a non-offset variable, or `ofvar` if it is an offset variable (i.e. referring to a local non-offset variable).
- `value(e)`: Value stored in the cell with address `e` (possibly `nil`).

- `arity(e)`: Arity of the constructor or variable stored at address `e`, the arity of a variable being `0`.
- `deref(e, S)`: The application of a substitution `S` to the expression representation at address `e`.
- `occurs-in(e1, e2)`: Checks whether a variable at address `e1` occurs-in the expression representation at address `e2`.

The algorithm consists of 16 cases given in 4 tables. Each case is characterised by `type(e1)`, `type(e2)` and the (dereferenced) expression representations at addresses `e1` and `e2`.

	type(e2) = cons value(e2) = s2/a2	type(e2) = novar value(e2) = nil
type(e1) = cons value(e1) = s1/a1	if value(e1) = value(e2) then R1 += arity(e1) R2 += arity(e2) else A := NU	S2 += (e2, deref(e1, S1)) R1 += arity(e1) if A = VR then A := SI if A = SG then A := OU
type(e1) = novar value(e1) = nil	S1 += (e1, deref(e2,S2)) R2 += arity(e2) if A = VR then A := SG if A = SI then A := OU	S2 += (e2, e1)

If both expressions are unbound variables then the variable at address `e2` is bound to that at address `e1` what avoids generating cyclic substitutions.

	type(e2) = ofvar deref(e2,S2) != nil	type(e2) = ofvar deref(e2,S2) = nil

type(e1) = cons
value(e1) = s1/a1
	unif(deref(e1, S1),
	     deref(e2, S2))
		de1 := deref(e1, S1)
		de2 := deref(e2, S2)
		if occurs-in(de2, de1)
		then A := NU
		else R1 += arity(e1)
		     S1 += (de2, de1)
		     S2 += (de2, de1)
		     A := OU

type(e1) = novar value(e1) = nil	S1 += (e1, deref(e2,S2)) if A = VR then A := SG if A = SI then A := OU	S1 += (e1, deref(e2,S2)) if A = VR then A := SG if A = SI then A := OU

If the representation at address `e1` starts with a constructor and the address `e2` is a bound variable, then the algorithm is recursively called. Otherwise, a binding is only generated if the occurs check fails. Unbound variables can be bound to any variable whether bound or unbound.

	type(e2) = cons value(e2) = s2/a2	type(e2) = novar value(e2) = nil
type(e1) = ofvar deref(e1,S1) != nil	unif(deref(e1, S1), deref(e2, S2))	S2 += (e2, deref(e1, S1)) if A = VR then A := SI if A = SG then A := OU

type(e1) = ofvar
deref(e1,S1) = nil
	de1 := deref(e1, S1)
	de2 := deref(e2, S2)
	if occurs-in(de1, de2)
	then A := NU
	else R2 += arity(e2)
	     S1 += (de1, de2)
	     S2 += (de1, de2)
	     A := OU
		S2 += (e2, deref(e1, S1))
		if A = VR then A := SI
		if A = SG then A := OU

The four cases above are symmetrical to the preceding four cases.

	type(e2) = ofvar deref(e2,S2) != nil	type(e2) = ofvar deref(e2,S2) = nil

type(e1) = ofvar
deref(e1,S1) != nil
	unif(deref(e1, S1),
	     deref(e2, S2))
		de1 := deref(e1, S1)
		de2 := deref(e2, S2)
		if occurs-in(de2, de1)
		then A := NU
		else A := OU
		     S1 += (de2, de1)
		     S2 += (de2, de1)

type(e1) = ofvar
deref(e1,S1) = nil
	de1 := deref(e1, S1)
	de2 := deref(e2, S2)
	if occurs-in(de1, de2)
	then A := NU
	else A := OU
	     S1 += (de1, de2)
	     S2 += (de1, de2)
		de1 := deref(e1, S1)
		de2 := deref(e2, S2)
		S1 += (de2, de1)
		S2 += (de2, de1)

Two offset variables pointing to bound variables result in a recursive call after applying substitutions `S1` and `S2`. Two offset variables only one of which points to an unbound variable require an occurs check. However, two offset variables both pointing to unbound variables make an occurs check unnecessary.

The time complexity of the algorithm is dominated by both the occurs check and the compatibility check both of which depend on the lengths of the expressions and the number of offset variables (ofvar-nb) bound to non-offset variables. Thus, the time complexity of the algorithm given above is in $O(\max(\text{length}(e_1), \text{length}(e_2)) \times \max(1, \text{ofvar-nb}(e_1) + \text{ofvar-nb}(e_2)))$.

To sum up, keeping track of the matching mode, as long as the expression prefixes traversed match, and dinstinguishing between locally bound, or offset, variables, and non-locally bound variables makes it possible to avoid unnecessary occurs check.

Further work will be devoted to a experimental comparison of the algorithm given above with formerly proposed unification algorithms.

References

[1] Dennis de Champeaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences*, 32(1):79–90, 1986.

[2] Gonzalo Escalada-Imaz and Malik Ghallab. A practically efficient and almost linear unification algorithm. *Artificial Intelligence*, 36(2):249—263, September 1988.

[3] Krystof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *Proceedings of KI 2009 – Advances in Artificial Intelligence, 32nd Annual German Conference on AI*, number 5803 in LNCS, pages 435–443. Springer, 2009.

[4] Alberti Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transaction on Programming Language Systems (TOPLAS)*, 4(2):258–282, April 1982.

[5] Michael Stewart Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[6] John Allan Robinson. A machine-oriented logic based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.