A Specification of the Unification Algorithm in CASL

Patrícia D. L. Machado[*]

LFCS - Division of Informatics,
University of Edinburgh, UK, pdlm@dcs.ed.ac.uk

DSC/COPIN - Federal University of Paraíba,
Campina Grande - PB - Brazil, patricia@dsc.ufpb.br

November 2, 2000

Abstract

Unification, which plays a central hole in theorem-proving, is the process of finding a common instance of two expressions, and if such an instance exists, the algorithm produces a substitution that yields that instance. The unification specification presented here is modelled on the one presented in [MW81] and is used in the author's PhD thesis to produce a case study on generating test oracles from structured algebraic specifications [Mac00].

Expression. A specification of expressions, namely EXPRESSION, is given as follows. An expression can be a constant, a variable or a function application to a list of expressions. The FUNCTION, VAR and CONST specifications model functions, variables and constants respectively.

spec FUNCTION =
  NAT
and
  STRING
then
  sort *function*;
  ops $declare \; : \; String \times Nat \to function$;
    $arity \; : \; function \to Nat$;

[*]Supported by CNPq - Brazilian Research Council

vars n : $String$;
i : Nat

- $arity(declare(n, i)) = i$

end

spec VAR =
CHAR
with
sort $Char \mapsto variable$
end

spec CONST =
CHAR
with
sort $Char \mapsto constant$
end

spec EXP =
sort $expression$;
end

spec EXPRESSION =
FUNCTION **and**
CONST **and**
LIST [EXP **fit sort** Elem ↦ expression] **and**
FINITESET [VAR **fit sort** Elem ↦ variable]
then
ops mk_c : $constant \to expression$;
mk_v : $variable \to expression$;
mk_f : $function \times List[expression] \to expression$;
$variables$: $expression \to FinSet[variable]$;
$variablesl$: $List[expression] \to FinSet[variable]$;

preds is_valid : $expression$;
is_validl : $List[expression]$;

vars c : $constant$;
v : $variable$;
f : $function$;
e : $expression$;
l : $List[expression]$

- $is\_valid(mk\_c(c))$
- $is\_valid(mk\_v(v))$
- $is\_valid(mk\_f(f, l)) \Leftrightarrow \#l = arity(f) \wedge is\_validl(l)$
- $is\_validl(nil)$
- $is\_validl(e :: l) \Leftrightarrow is\_valid(e) \wedge is\_validl(l)$
- $variables(mk\_c(c)) = \{\}$

- $variables(mk\_v(v)) \ = \ set(v)$
- $variables(mk\_f(f, l)) \ = \ variablesl(l)$
- $variablesl(nil) \ = \ \{\}$
- $variablesl(e :: l) \ = \ (variables(e)) \ union \ (variablesl(l))$

hide
$variablesl, is\_validl$
end

where List[Elem] and FiniteSet[Elem] are generic specifications of sets and lists respectively with the usual operations. An expression is valid when functions are applied to the correct number of arguments.

Substitution. Substitution is an operation that replaces specific variables of an expression by other expressions. The Substitution specification is presented below as an extension of Expression.

spec Substitution =
Expression
then
sort $substitution$;
ops $empty \ : \ substitution$;
$dom \ : \ substitution \rightarrow FinSet[variable]$;
$rng \ : \ substitution \rightarrow FinSet[variable]$;
$add\_s \ : \ variable \times expression \times substitution \rightarrow substitution$;
$apply \ : \ expression \times substitution \rightarrow expression$;
$applyl \ : \ List[expression] \times substitution \rightarrow List[expression]$;
$compose \ : \ substitution \times substitution \rightarrow substitution$;

vars $x \ : \ variable$;
$e \ : \ expression$;
$s \ : \ substitution$

- $dom(empty) \ = \ \{\} \ \wedge \ rng(empty) \ = \ \{\}$
- $\neg \ (x \ elemOf \ dom(s)) \ \wedge \ \neg \ (e \ = \ mk\_v(x)) \ \Rightarrow$
  $dom(add\_s(x, e, s)) \ = \ (set(x)) \ union \ (dom(s)) \ \wedge$
  $rng(add\_s(x, e, s)) \ = \ (variables(e)) \ union \ (rng(s))$

vars $c \ : \ constant$;
$x, y \ : \ variable$;
$f \ : \ function$;
$e \ : \ expression$;
$s, s' \ : \ substitution$;
$l \ : \ List[expression]$

- $apply(e, empty) \ = \ e$
- $apply(mk\_c(c), s) \ = \ mk\_c(c)$
- $apply(mk\_v(x), add\_s(x, e, s)) \ = \ e$

- $\neg\,(x = y) \Rightarrow apply(mk\_v(y), add\_s(x,e,s)) = apply(mk\_v(y), s)$
- $apply(mk\_f(f,l), s) = mk\_f(f, applyl(l,s))$
- $applyl(nil, s) = nil$
- $applyl(e :: l, s) = (apply(e,s)) :: (applyl(l,s))$
- $apply(e, compose(s,s')) = apply(apply(e,s), s')$

vars $s, s', s'' : substitution$
- $compose(s, empty) = s \wedge compose(empty, s) = s$
- $compose(compose(s,s'), s'') = compose(s, compose(s', s''))$

hide
applyl
end

where the second axiom says that if x is not in $dom(s)$ and e is an expression distinct from $mk\_v(x)$, then the result of adding the replacement "e for x" to the substitution s is also a substitution $add\_s(x,e,s)$ such that the conditions showed there hold.

Unification. The UNIFICATION specification is given as follows as an extension of the SUBSTITUTION specification. A substitution is a unifier of two expressions if applying this substitution to both expressions yields the same result. If two expressions are unifiable, they have a most general and idempotent unifier.

spec UNIFICATION =
SUBSTITUTION
then
op $unify : expression \times expression \to substitution$;
preds $is\_moregeneral : substitution \times substitution$;
$is\_idempotent : substitution$;

axiom $\forall\, e, e' : expression \bullet$
$(\exists\, s : substitution \bullet apply(e,s) = apply(e',s)) \Rightarrow$
$apply(e, unify(e,e')) = apply(e', unify(e,e')) \wedge$
$(\forall\, s' : substitution \bullet$
$apply(e,s') = apply(e',s') \Rightarrow$
$is\_moregeneral(unify(e,e'), s')) \wedge$
$is\_idempotent(unify(e,e'))$
axiom $\forall\, s, s' : substitution \bullet$
$is\_moregeneral(s,s') \Leftrightarrow$
$(\exists\, r : substitution \bullet s' = compose(s,r))$
axiom $\forall\, s : substitution \bullet$
$is\_idempotent(s) \Leftrightarrow dom(s)\ intersection\ rng(s) = \{\}$
hide
$is\_moregeneral, is\_idempotent$
end

References

[Mac00] Patricia D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem.* PhD thesis, LFCS - Division of Informatics, University of Edinburgh, 2000.

[MW81] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.