Program Transformation and Theorem Proving As Constraint Propagation

Jean-Luc Dormoy
Electricité De France R&D Center
1, avenue du général De Gaulle
92141 Clamart Cedex - FRANCE
Jean-Luc.Dormoy@der.edf.fr
Tel: 33 1 47 65 59 06
Fax: 33 1 47 65 54 28

Content Areas : Automatic Programming, Automated Reasoning.
Keywords: Automatic Programming, Algorithm design, Specification language, Knowledge-based system, Specification refinement, Program transformation.

Abstract: While carrying out the design and implementation of the automatic program generator Descartes, we have been conducted to develop new means of reasoning on extended first-order expressions. Descartes needs to prove theorems on expressions, depending on their context within the specification it receives as input. We have defined a new weak form of theorem proving, which applies to sets, logical expressions, and real numbers.
The method is based on the definition of abstract operators which generalize the set, logical and real operators, in particular a partial order relation, which instanciates as inclusion, logical implication, or the conventional real order relation. For any given interesting expression e, intervals of expressions including e are computed. These intervals can be propagated to subexpressions, making it possible to prove new theorems at the meta-level, and to process simplifications of expressions which depend on their context. Various examples of application of this technique are given throughout the paper.

1 Introduction

Knowledge-based program automatic synthesis has been the concern of active research for more than ten years. Several categories of knowledge that should be encapsulated have been identified. Examples are folding and unfolding of function calls, recursion removal, formal differentiation, evaluation of first-order queries. Using all these knowledge chunks means transforming step by step the initial expression of the specification to reach a final form considered as implementing a good algorithm. So, they all generate new expressions, which need not be in their simplest form. In turn, new transformations can often be applied, or interesting clichés recognized, only if the specification is properly simplified. Simplifying expressions is thus a crucial requirement for an efficient autonomous program generator.

Usually, simplifications are performed by rewriting rules. Descartes possesses a set of about 300 such rewriting rules. Nevertheless, our experience in automatic program synthesis shows that they are necessary, but not sufficient. Other simplifications should be performed, which depend on the context within which an expression occurs.

This paper describes a propagation method we have designed, which is similar in its spirit to interval propagation for real expressions.

Section 2 provides some simple motivating examples. Section 3 defines abstract operators and intervals of expressions, and their use for some of the examples. Section 4 describes the propagation of local theorems, and their use in rewriting. Section 5 does the same for contextual theorems. Section 6 describes how local and contextual theorems stemming from propagation can be combined to deduce new theorems. Section 7 shows how propagation can "jump over" recursion by means of induction. Section 8 suggests some directions in future work. Finally, Section 9 concludes.

2 Some examples

The input specification language of Descartes is based on functions and sets. We do not intend to precisely describe here its syntax not its range. Instead, we shall use examples, which intuitively illustrate the main features of the Descartes language.

Let's take simple examples of Descartes legal expressions.

(i) $((F \approx \{a\}) - \{b\}) \approx \{a\}$

(ii) $\forall x \forall y \ \ (x,y) \in E \Rightarrow [\forall x' \forall y' \ \ (x',y') \in E \Rightarrow [x=x' \Rightarrow y=y']]$

(iii) $x \in E$ and $y \in E \approx \{x\}$

(iv) $x \in E$ or $y \in E - \{x\}$

(v) $F \subset E$ and $y \in E \approx (F-G)$

(vi) If $x \notin F$ Then $f(E-(F-\{x\}))$ Else $f(F \approx \{x\})$

E, F, G are sets, f is a function, a, b are constants, and the other operators have the usual meaning. It is clear that these expressions can be simplified as:

(i)' $(F - \{b\}) \approx \{a\}$
(ii)' $\forall x \forall y \ \ (x,y) \in E \Rightarrow [\forall y' \ \ (x,y') \in E \Rightarrow y=y']$
(iii)' $x \in E$ and $y \in E$
(iv)' $x \in E$ or $y \in E$
(v)' $F \subset E$ and $y \in E$
(vi)' If $x \in F$ Then $f(F)$ Else $f(E-F)$

We intend to show how this kind of simplification is possible.

3 Intervals of expressions

3.1 Abstraction of logical, set and arithmetic operators

We define abstract operators ˜, ⟨, $\vee$, $\wedge$, --, π, 0, 1 and σ, generalizing set, logical and real operators, in the following way ($\equiv$ means syntactic identity):

Logical expressions:

e ˜ f	$\equiv$	$e \Leftrightarrow f$
e ⟨ f	$\equiv$	$e \Rightarrow f$
$e \vee f$	$\equiv$	e or f
$e \wedge f$	$\equiv$	e and f
-e	$\equiv$	$\neg e$
$\pi(e,x)$	$\equiv$	$\exists x \ \ e$
0	$\equiv$	False
1	$\equiv$	True
$\sigma(e,\{(x_1,t_1),\ldots,(x_k,t_k)\})$	$\equiv$	e where for all $j=1,\ldots,k$, term t_j is substituted for variable x_j

Sets:

E ˜ F	$\equiv$	$E = F$
E ⟨ F	$\equiv$	$E \subset F$
$E \vee F$	$\equiv$	$E \approx F$
$E \wedge F$	$\equiv$	E −E ≡
$\pi(E,i)$	$\equiv$	Projection of set E orthogonal to its ith component (E is the subset of a cartesian product)
0	$\equiv$	Ø (the empty set)
1	$\equiv$	1 (the full set)
$\sigma(E,\{(i_1,t_1),\ldots,(i_k,t_k)\})$	$\equiv$	Projection of the tuples of set E orthogonal to $i_1,\ldots i_k$ such that for all $j=1,\ldots,k$, their jth component is equal to term t_j

Arithmetic expressions:

$a \sim b$	$\equiv$	$a = b$
$a \langle b$	$\equiv$	$a < b$
$a \vee b$	$\equiv$	$\max(a,b)$
$a \wedge b$	$\equiv$	$\min(a,b)$
$--a$	$\equiv$	$-a$
0	$\equiv$	$-$
1	$\equiv$	$+$

π and σ are left undefined for arithmetic expressions.

This generalization of set and logical operators is well-known, for example in database theory. Including arithmetic expressions is natural, once the properties of $\sim$, $\langle$, … are established. In particular, $\langle$ is an order relation, $\vee$ and $\wedge$ are commutative, associative and idempotent, etc. We also have other relations, such as $-(x \vee y) = (-x \wedge -y)$, or $(x \sim y) = ((x \langle y) \wedge (y \langle x))$. We assume that all these properties are well-known.

The introduction of a "full set" 1 (the complementary of the empty set) may seem strange with respect to orthodox set theory (the set of all sets!?). Indeed, there is no difficulty in defining it here, much in the same way as the set of real numbers can be supplemented with + and - , as we only consider a simple naive finite set theory.

3.2 The use of intervals of expressions: two examples

By means of the order relation , we can compute and propagate intervals of expressions. To do this, we define S(e) (resp. I(e)) for an expression e as a set of expressions f such that $e \langle f$ (resp. $f \langle e$). Obviously, all expressions f greater than e will not be computed, only "interesting" expressions will be, according to the context. This, together with rewriting and normalization rules, can lead to interesting simplifications.

Let's consider as an example Expression (i),

(i) $((F \approx \{a\}) - \{b\}) \approx \{a\}$

Such an expression is normalized in the following sequence of expressions to reach expression (i)". The rules used in this transformation are the normalization of set difference E - F --> E $((\Phi \approx \{a\})\ \{a\}$

$((F \vee \{a\}) \wedge -\{b\}) \vee \{a\}$

(i)" $(F \wedge -\{b\}) \vee (\{a\} \wedge -\{b\}) \vee \{a\}$

Now, we know that $\{a\} \subset \{a\}$. Using our abstract vocabulary, this means $\{a\} \langle \{a\}$, or "$\{a\}$" $\in$ S ("$\{a\}$") (we quotate expressions when necessary in order to distinguish the basic and meta-level set languages). From this, and properties of $\wedge$, we can deduce that "$\{a\}$" $\in$ S("$\{a\} \wedge -\{b\}$"). Finally, using the generalized rewriting rule

$[p \in S(E), p \in I(F)]$ $E \vee F$ --> F

we can deduce that (i)" is equal to expression (i)'"

(i)'" $(F \wedge -\{b\}) \vee \{a\}$

Let's now consider expression (iii).

(iii) $x \in E$ and $y \in E \approx \{x\}$

While in the previous example, expression (i) was rewritten for reasons intrinsic to its content, this is not possible here, since expression $E \approx \{x\}$ and E are not equal in general. So, we need to state that $x \in E$ is true at the level of the whole expression, or in its *context*. Then, $E \approx \{x\}$ can be simplified as E. We shall describe in Section (v) how this is formalized.

The other examples given in Section 2 can be treated in a similar way.

3.3 Aims and constraints of interval propagation

We intuitively feel that these ideas can lead to a general method for simplifying expressions. Its main three features are:

- Rewriting and simplifying expressions cannot be achieved in general by gross rewriting rules. Local theorems stemming from the expressions themselves must be proved first.

- Simplification of expressions can depend on the context of the expressions, that is the expressions of which they are subexpressions. So, local theorems must be propagated up and down in the expression tree. “Down” means considering contextual theorems.

- The method must remain tractable. In particular, the number of local and contextual theorems computed and propagated must remain a polynomial function (preferably linear) of the number of nodes in the expression tree.

4 Propagation of local theorems

Propagating local theorems is achieved by a set of rules. These rules are not the most general ones, as they must not produce too many theorems.

4.1 Basic propagation rules and "boot" of propagation

The first set of rules concerns the basic operators $\vee$, $\wedge$, - and π.

$p \in S(e) \rightarrow p \in S(e \wedge f)$ (S1)
$p \in S(e), p \in S(f) \rightarrow p \in S(e \vee f)$ (S2)
$p \in S(e) \rightarrow -p \in I(-e)$ (S3)
$p \in S(e) \rightarrow \pi(p,x) \in S(\pi(e,x))$ (S4)

$p \in I(e) \rightarrow p \in I(e \vee f)$ (I1)
$p \in I(e), p \in I(f) \rightarrow p \in I(e \wedge f)$ (I2)
$p \in I(e) \rightarrow -p \in S(-e)$ (I3)
$p \in I(e) \rightarrow \pi(p,x) \in I(\pi(e,x))$ (I4)

The reader can check the validity of these rules. Rules (Si) and (Ii) are dual of each other through operator -.

The propagation must start from a "boot" of S and I. The boot is made up of specific theorems $e \in S(e)$ and $e \in I(e)$. First of all, when $p \in S(e)$ or $p \in I(e)$ is to be stated, e must be an expression occurring in the specification, and p must be as "simple" as possible. We accept as simple expressions p the expressions recursively defined as being: an atomic expression; $t_1 = t_2$, $t_1 \in t_2$, $t_1 \subset t_2$, $t_1 = t_2$, $\neg e$, $\exists x\ e$, sets in comprehension $\{t_1 \mid t_2\}$, where t_1, t_2 and e are simple expressions; enumerated sets $\{t_1,\ldots,t_n\}$, where $t_1,\ldots t_n$ can be any terms; $t_1 = t_2$, $\exists x\ e$, $f(t_1)$, where f is a recursive function, with no condition on t_1, t_2 and e. It is clear that previous propagation rules are respectful of this definition of simple expressions. So, we just have to pay attention to include only simple expressions in the boot.

Indeed, we fill in the boot with theorems $e \in S(e)$ and $e \in I(e)$ for all simple expressions e occurring in the specification.

4.2 Bridge rules

Our abstract vacabulary unifies logical, set and arithmetic expressions. Nevertheless, we need to be able to switch local theorems from one form to the other. For example, if the specification contains the expression $x \in e$, where e is a complex expression, and if we know that $p \in S(e)$ (i.e. that e is included in p), then one should deduce that "$x \in p$" $\in S(x \in e)$. So, we need "bridge rules" to switch in both directions from theorems on sets to theorems on logical expressions. These rules are

$$
\begin{array}{ll}
p \in S(e) & \rightarrow \quad \text{"}t \in p\text{"} \in S(t \in e) \\
p \in I(e),\ q \in S(f) \rightarrow & \text{"}p \langle q\text{"} \in S(e \langle f),\ \text{"}p \langle q\text{"} \in S(e=f) \\
p \in S(C) & \rightarrow \quad \text{"}\{a \mid p\}\text{"} \in S(\{a \mid C\})
\end{array}
$$

plus similar dual rules.

4.3 The complexity of basic propagation

A program specification is an expression e. Syntactically, it is a tree, or more precisely a directed acyclic graph (dag), as syntactically equal expressions occurring at different places can be represented by a single node. The complexity of propagation should be measured in terms of the *number of nodes* N in this dag.

It is clear from what preceeds that the size of the boot is linear in N. Moreover, for any expression e occurring in the specification, the number of expressions added to its S and I sets is bounded by the *sum* of the similar numbers for its child expressions. So, the total number of theorems computed and propagated is at most N^2 (and much less in a practical way). The propagation itself takes a time limited by this quantity, provided that the data structures for representing expressions are suitable.

4.4 Limitations of propagation

Limiting the propagation to a polynomial complexity is strongly desirable. This means that propagation can be left in principle "uncontrolled", while providing acceptable running times. Otherwise, a serious control problem would rise, which we did not intend to tackle. The definition of "simple expressions" susceptible of belonging to a S(e) or I(e) has been chosen with this in mind.

In particular, this explains why we implemented a limited version of the propagation rules. The main limitation lies in rules (S2) and (I2). The general form of these rules would be:

$p \in S(e), q \in S(f) \quad \rightarrow \quad "p \vee q" \in S(e \vee f) \qquad$ (S'2)
$p \in I(e), q \in I(f) \rightarrow \quad "p \wedge q" \in I(e \wedge f) \qquad$ (I'2)

It is clear that, if we authorize these rules, then the number of expressions in $S(e \vee f)$ becomes a *product* of the number of expressions in S(e) and S(f), so leading to an exponential propagation process.

This does not mean that rules (S'2) and (I'2) are unuseful. It turns out that they are rarely so, and we could not imagine a proper control that would make them acceptable.

Nevertheless, we have implemented particular versions of these rules, when p and q are *enumerated* expressions. For example, if p and q are the set expressions {a,b,c} and {c,d}, and if $p \in I(e)$ and $q \in I(f)$, it is very informative to know that "{c}" $\in$ I(e). This particular case does not spoil the polynomial behavior of propagation.

4.5 Theorem-dependant rewriting rules

As shown in the examples, the goal of interval propagation is to make simplifications possible. This is achieved by rules, the conditions of which have two parts: a condition on local theorems (in brackets), and the pattern of the expression to be simplified. The consequent is the rewritten expression.

Here is a set of such rules. Dual rules (obtained by properly applying the operator -) can be imagined from them. The first set is essentially based on the Boolean lattice properties of 〈, ∧, ∨ and -. We have grouped the rules in subsets of more and more general rules (specialized rules are useful for improving rewriting time).

$[0 \in S(e)] \quad e \quad \rightarrow \quad 0$
$[p \in S(e), -p \in S(e)] \quad e \quad \rightarrow \quad 0$
$[p \in S(e), q \in S(e), p \in I(f), -q \in S(f)] \quad e \quad \rightarrow \quad 0$

$[p \in S(e), p \in I(e), p?e] \quad e \quad \rightarrow \quad p$

$[p \in I(e), p \in S(f)] \quad e \vee f \rightarrow \quad e$
$[\forall\, i \in [1,n] \;\; \exists\, p_i \in I(f) \;\; p_i \in S(e_i)]$
$\quad (e_1 \wedge \ldots \wedge e_n) \vee f \rightarrow \quad e_1 \wedge \ldots \wedge e_n$

$$[\forall\, i\in[1,n] \quad \exists\, p_i \quad \exists\, j\in[1,m] \quad p_i\in S(e_i) \text{ and } p_i\in I(f_j)]$$
$$(e_1 \wedge \ldots \wedge e_n) \vee (f_1 \wedge \ldots \wedge f_m) \quad \rightarrow \quad e_1 \wedge \ldots \wedge e_n$$

The second set of rules deduces the consequences of equalities. Similar rules also apply to sets $\{f(x) \mid C(x)\}$, but we have not mentioned them.

$$[\text{"}x = t\text{"} \in S(e)] \quad \pi(e,x) \quad \rightarrow \quad \sigma(e,\{(x,t)\})$$
$$[\pi(\text{"}x = t\text{"},u) \in S(e)] \quad \pi(e,x) \quad \rightarrow \quad \pi(\sigma(e,\{(x,t)\}),u)$$

The first previous rule is the key step in simplifying expression (ii) of Section 2.

5 Contextual theorems

5.1 Definition of contextual theorems

Theorems deduced by the propagation rules so far described are intrinsic to the expressions they mention. This is not sufficient to deduce simplifications, as shows example (iii) (cf. Section 3.2).

We need to introduce a new kind of theorem, contextual theorems. We define new meta-sets SC(e,a,A) and IC(e,a,A), where e is a subexpression of a, and a is a child expression of A. We write A(t) the expression where t is substituted for a given occurrence of e below a. SC and IC must have the following property:

if $p\in SC(e,a,A)$ then $A(e)$ is equivalent to $A(e \wedge p)$
if $p\in IC(e,a,A)$ then $A(e)$ is equivalent to $A(e \vee p)$

If p and e are logical expressions, $p\in SC(e,a,A)$ means that p can be considered as being true "below" a. If they are set expressions, $p\in SC(e,a,A)$ means that $e \subset p$ can be considered true below a. For example, in

(ii) $x\in E$ and $y\in E\approx\{x\}$

and if A denotes the whole expression and a the expression "$y\in E\approx\{x\}$", we have

"$x\in E$" $\in SC(a,a,A)$

and, according to our definition, we also have

"$\{x\}$" $\in IC(E,a,A)$

SC and IC actually are generalizations of S and I: if $p\in S(A)$, then A is equivalent to "$A \wedge p$".

5.2 Establishing contextual theorems

5.2.1 From local to contextual theorems

There are several ways to establish contextual theorems. The first one is given by the following rules, where a is a child expression of A.

$$p \in S(A),\ p \notin S(a) \quad \rightarrow \quad p \in SC(a,a,A)$$

$p \in I(A), p \notin I(a) \quad \rightarrow \quad \neg p \in SC(a,a,A)$

The first rule is obvious. One would expect as second rule

$p \in I(A), p \notin I(a) \quad \rightarrow \quad p \in IC(a,a,A)$

which is valid. Nevertheless, the form given to the second rule is more informative within a context of automatic programming, where expressions have to be sooner or later evaluated. Let's consider expression (iv), denoted by B:

(iv) $x \in E$ or $y \in E-\{x\}$

It is clear that this expression is equivalent to

(iv)' $x \in E$ or $y \in E$

Indeed, if $x \in E$, (iv) is satisfied. Otherwise, we have to check that $y \in E-\{x\}$, and, as we can assume that $x \notin E$, this is equivalent to $y \in E$. This is why our second rule will infer

$"\neg x \in E" \in SC("y \in E-\{x\}","y \in E-\{x\}",B)$

which is a step of previous reasoning. So, to sum up, while the first rule means that p can be considered *true* below a and A, the second rule states that p can be considered *false* below a and A.

These rules explain why we need to introduce A's child expression a. The informal contextual theorem "p is true (resp. false) within A" should not be applied to a child expression b of A from which this property comes from. Otherwise, context-dependant rewriting rules could apply to b, and destroy truth or falsity p.

5.2.2 Bridge rules

As for local theorems, there are bridge rules linking sets and logic. For example, we have the rules

$"t \in E" \in SC(a,a,A) \quad \rightarrow \quad \{t\} \in IC(E,a,A)$

$"\neg t \in E" \in SC(a,a,A) \quad \rightarrow \quad -\{t\} \in SC(E,a,A)$

We omit similar rules involving expressions p mentioning inclusion, etc.

5.2.3 If-Then-Else rules

Contextual theorems apply to If-Then-Else expressions. If A is the expression "If a Then b Else c", we have:

$p \in S(a) \quad \rightarrow \quad p \in SC(b,b,A)$

$p \in I(a) \quad \rightarrow \quad \neg p \in IC(c,c,A)$

5.2.3 Horn clauses deductions

Contextual theorems can be propagated in more complex cases. For example, consider the expression

$\exists w \quad [\exists b \ (w,b) \in D \text{ and } (v,w,a,b) \in C]$ and

$[\forall b' \ (w,b') \in D \text{ and } (v,w,a,b') \in C \Rightarrow (w,b') = (v_0,a_0)]$

where v, a, v_0, a_0, C and D are parameters (free variables). It is normalized as

$\exists w \quad [\exists b \ \ (w,b)\in D \ \text{ and } \ (v,w,a,b)\in C] \ \text{ and}$
$\qquad [\neg\exists b' \ \ (w,b')\in D \ \text{ and } (v,w,a,b')\in C \ \text{ and } \ \neg(w,b') = (v_0,a_0)]$

It is visible in the first form that the "b", which must exist, must also check the second subexpression, and so verify $(w,b) = (v_0,a_0)$. So, we necessarily have $w = v_0$ and $b = a_0$, and, finally, the expression can be rewritten

$(v_0,a_0)\in D \ \text{ and } \ (v,v_0,a,a_0)\in C \ \text{ and}$
$\qquad [\forall b' \ \ (v_0,b')\in D \ \text{ and } (v,v_0,a,b')\in C \Rightarrow b' = a_0]$

This reasoning is made possible by using the following rule:

"$\pi(e_1\wedge\ldots\wedge e_n,y)$" $\in$ S(A),
"$-\pi(f_1\wedge\ldots\wedge f_n\wedge g,x)$" $\in$ S(A),
for all $i\in[1,n]$ there exists $p_i \in S(e_i)$ such that $\sigma(p_i,\{(y,x)\}) \in I(f_i)$
$\qquad\qquad \rightarrow \quad \neg\sigma(g,\{(x,y)\}) \in SC(e_1\wedge\ldots\wedge e_n,e_1\wedge\ldots\wedge e_n,A)$

provided that g is simple (in which case $\neg\sigma(g,\{(x,y)\})$ is also simple).

Indeed, in this rule, the second expression is used as a clause applied to the first expression, and as the "tail litteral" g must be simple, it is something resembling a Horn clause.

Once this rule has been fired, the reader can check that the already mentioned theorem-dependant rewriting rules, together with simpler rewriting rules, yield the desired result.

There are other similar forms of this rule that we omit.

Remark: The reader can check that applying this rule is polynomial with respect to the size of expression A.

5.3 Context-dependant rewriting rules

Most simplification rules described in Section 4.5 can be adapted to apply to contextual theorems. For example, the rule

$[p \in I(e), p \in S(f)] \qquad e \vee f \rightarrow \quad e$

becomes

$[p \in IC(e,a,A), p \in SC(f,a,A)] \qquad e \vee f / A \quad \rightarrow \quad e / A$

which means that, under the prescribed conditions, $e \vee f$ should be rewritten as e *in its occurrences as a subexpression of* A. Other rules can be imagined in a similar way.

6 Meta-level theorem proving

Assume for example that we have deduced that "$F\subset E$" and "$E\subset F$" both belong to S(e). It is clear that we should deduce that "E=F" belongs to S(e). Indeed, if p and q both belong to S(e), then "$p \wedge q$" implicitely also belongs to S(e) (it does not actually belong to S(e), since it is not simple). In the same way, if p and q both belong to

I(e), then "p $\vee$ q" implicitely belongs to I(e). This can be extended to contextual theorems. New interesting local or contextual theorems could be deduced from these considerations.

To do this, we have implemented some rules achieving in a limited way this kind of meta-level theorem proving. We have designed them in the same spirit as our propagation method. There are several groups of such rules. For each group we mention a single rule (rules for I(e) can be deduced by properly using the - operator, and the rules can be extended to contextual theorems).

Transitivity of á:

"p〈q" $\in$ S(e), "q〈r" $\in$ S(e) $\rightarrow$ "p〈r" $\in$ S(e)

Antitransitivity of á:

"p〈q" $\in$ S(e), "q〈p" $\in$ S(e) $\rightarrow$ "p=q" $\in$ S(e)

"p〈q" $\in$ S(e) and "q〈p" $\in$ S(e) are then removed from S(e).

Partial evaluation of "p Ù q" and "p Ú q":

If p and q are enumerated sets made up of known constants:

p $\in$ S(e), q $\in$ S(e) $\rightarrow$ (p ωηερε (π

Weakened local theorems:

If p="{p_1,…,p_n}" and q="{q_1,…,q_m} are enumerated sets:

p $\in$ S(e), q $\in$ S(e) $\rightarrow$ "{p_1,…,p_n,q_1,…,q_m}" $\in$ S(e)

Equality as an equivalence relation:

For this, we define two meta-operators, AllEq (All Equal) and OneD (One Different) defined on a set of expressions by

AllEq({p_1,…,p_n}) iff for all i,j$\in$[1,n] $p_i = p_j$

OneD({p_1,…,p_n}) iff there exists i,j$\in$[1,n] $\neg p_i = p_j$

Then, when "$t_1 = t_2$" $\in$ S(e), "$t_1 = t_2$" is transformed into AllEq({t_1,t_2}), and the "AllEq" expressions having one element in common are merged. The same is done for OneD and I(e).

7 Propagation through recursion

So far we have described various ways of propagating local and contextual theorems within first-order expressions. The Descartes language is functional, and so can mention recursive functions. Our experience shows that it can be crucial to propagate theorems through recursion.

Recursion can be complex. Nevertheless, the Descartes knowledge base possesses some knowledge, based on clichés, for removing recursion. This means that, whenever possible, complex recursion patterns will be transformed into tail-recursive equivalent expressions. Descartes can also remove cross-recursion. So, we have limited our presentation of theorem propagation to tail-recursive functions (though the results could be extended to more general patterns).

Let's consider a general pattern of tail-recursion:

```
    f(x)   =     If  a(x)  Then  b(x)
                 Else  f(θ(x))
```

where a, b and θ are functions not calling f.

Basically, theorem propagation through recursion is based on a limited use of inductive proofs. We have implemented several forms of reasoning. Firstly, assume that regular propagation has deduced a (local or contextual) theorem $P(\theta(x))$. Then, if f is called somewhere in the specification with an argument e which also verifies $P(e)$, one can assume that $P(x)$ is true within the context of this call.

For example, if we consider the function

```
g(D,T)      =    If  a(D,T)  Then  b(D,T)
                 Else  g(D - θ(D,T),θ(D,T))
```

where $\theta(D,T)$ is simple, then the local theorem "$-\theta(D,T)$" $\in$ $S(D - \theta(D,T))$ is deduced. If g is called somewhere in the specification as $g(D_0 - u(D_0),u(D_0))$, and if $u(D_0)$ is simple, then the theorem "$-u(D_0)$" $\in$ $S(D_0 - u(D_0))$ is also deduced. From this, Descartes deduces that "$-T$" $\in$ $S(D)$ is a contextual theorem within this particular calling of the function.

Now, this new theorem can be in turn propagated. If, for example

$$\theta(D,T) = \{d \mid d \in D \text{ and } \exists d' \; R(d,d') \text{ and } d' \in D \text{ and } d' \notin T\}$$

where R is any relation, then, within the function call, this expression can be simplified as

$$\theta(D,T) = \{d \mid d \in D \text{ and } \exists d' \; R(d,d') \text{ and } d' \in D\}$$

Another form of propagation through recursion occurs when a theorem $P(x,\theta(x))$ is proved and P is reflexive and transitive. It is in particular the case whan P is the theorem $x \in S(\theta(x))$ or $x \in I(\theta(x))$. Then, $P(e,x)$ is true within the context of a call $f(e)$. In particular, this can make it possible to prove new theorems on the final result $b(x)$ of f. For example, if $b(x) \equiv x$, then the theorem $P(f(e),e)$ is deduced.

8 Perspectives and future work

8.1 Other uses of the results of propagation

The results of propagation make it possible to generate efficient programs from queries, and in particular to select proper indices of data. We shall only give the following example. Consider the set mentioned in a specification

$$\theta(D,T,C) = \{(v,a) \mid (v,a) \in D \text{ and } \exists w \exists b \; (w,b) \in T \text{ and } (v,w,a,b) \in C\}$$

and assume that $T \in I(D)$ and $T \in I(\{(w,b) \mid \exists v \exists a \; (v,w,a,b) \in C\})$ have been deduced. Then, if it is necessary to generate a program which computes the set $\theta(D,T,C)$, a good algorithm is

```
S <-- $\emptyset$
For all $(w,b) \in T$ do
    For all $(v,a)$ such that $(v,w,a,b) \in C$ do
        If $(v,a) \in D$ Then S <-- $S \approx \{(v,a)\}$
```

First, it is better to have smaller sets for loop domains, so it is interesting to have the loop on $(w,b) \in T$ first. Second, the index $i(w,b) = \{(v,a) \mid (v,w,a,b) \in C\}$ will speed up the second loop, and Descartes decides that it should be constructed and maintained in the program.

Descartes contains knowledge, which performs this kind of optimization when eventually generating the algorithm.

8.2 Automatic design of context-dependant simplification

All the propagation, rewriting and meta-level theorems described here can be deduced from a common knowledge core on basic properties of set and logical expressions. So, it could be interesting to design a meta-level system, which would automatically generate them from this common core. This would in particular ensure some completeness in their design.

Moreover, this approach could lead to interesting enhancements. As we have suggested, specialized versions of our rules have been implemented to speed up the process. This specialization could be automatized and extended through some particular kind of learning.

Let's consider for example the rewriting rule

$[p \in I(e), p \in S(f)] \quad e \vee f \rightarrow \quad e$

This rule could be considered as a meta-rule

$p \in I(e), p \in S(f) \equiv> \quad (e \vee f \rightarrow \quad e)$

where, when $p \in I(e)$ and $p \in S(f)$ have been deduced in a particular situation, a new rewriting rule $e \vee f \rightarrow e$ is to be added to the system. Obviously, p, e and f must be generalized to make this new rule interesting. Moreover, some filter must be designed to discard rewriting rules that finally turn out to be unuseful. These ideas are close to the theorem discoverer designed by Pitrat [Pitrat, 66], the theorem prover Muscadet [Pastre, 89], to EBL (e.g. in Prodigy), and to the chunking mechanism of Soar.

8.3 Extending theorem-proving capabilities

Another interesting direction is to extend our propagation method beyond its "polynomial constraints" in time. For example, our system is unable to simplify the expression

$A \subset B \approx C$ and $A \subset A1 \approx A2$ and A1 ιντο

$A \subset B$ and $A \subset A1 \approx A2$ and A1

Τηερε αρε ϖαριουσ ωαψσ το σολϖε τηε πρεϖιουσ εξαμπλε, ανδ μορε γενερα λλψ το σιμπλιφψ εξπρεσσιονσ ινϖολϖινγ σετσ ωιτη νο οτηερ ϖαριαβλεσ. Η οωεϖερ, τηε γενεραλ χασε οφ φιρστ−ορδερ εξπρεσσιονσ ισ νοτ οβϖιουσ.

9 Conclusion

We have presented a propagation method making it possible to rewrite set and logical expression similar to interval propagation for real expressions. The propagation is based on the computation of intervals of expressions to which a given expression must belong, yielding local and contextual theorems. Propagation mainly occurs within first-order expressions, but can also "jump over" recursion. Meta-level theorem proving rules have also been designed, which extend the set of local and contextual theorems deduced by propagation. Rewriting rules mentioning in their condition part local or contextual theorems can then simplify the expressions. Finally, the whole process makes it possible to reach non trivial simplifications, while remaining polynomial in time and space.

References

[Barstow, 79] *Knowledge-based program construction*. Elsevier North Holland, New-York.

[Burstall, Darlington, 76] *A system which automatically improves programs*. Acta Inf. 6. pp 41-60.

[Burstall, Darlington, 77] *A transformation system for developing recursive programs*. J. ACM 24, 1, pp 44-67.

[Dague, 93] Numeric reasoning with Relative Orders of Magnitude. AAAI’93, Washington DC.

[Garijo, 78] *GPFAR 2: A program automatic synthesizer for the optimized computation of recursive functions*. PhD Dissertation Paris 6 University.

[Ginoux B., 91] *Knowledge-based automatic algorithm generation from very high-level declarative specifications: The COGITO system*. PhD dissertation, Paris IX University.

[Ginoux, Lagrange, 89a] *An expert system approach to program synthesis*. AAAI Spring Symposium Series 1989, Artificial Intelligence and Software Engineering, Standford, Ca. USA.

[Ginoux, Lagrange, 89b] *Synthesis of simple programs which handle complex data.*, IJCAI'89 Workshop on Automating Software Design, Detroit, Mich. USA.

[Kant, Barstow, 81] *The refinement paradigm : The interaction of coding and efficiency knowledge in program synthesis*. IEEE Trans. Softw. Eng. 7, 458-471.

[Laird, Newell Rosenbloom, 87] *Soar: An architecture for general intelligence*. AI J. 33, 1, pp1-64.

[Lhomme, 93] *Consistency techniques for numeric CSPs*. IJCAI’93, Chambéry, France.

[Minton et al., 89] Minton, Carbonell, Knoblock, Kuokka, Etzioni, Gil. *Explanation-based learning: A problem solving perspective*. AI J. 40, pp 63-118.

[Mostow, 91] *A transformational approach to knowledge compilation*. In Lowry & McCartney, Eds, *Automating software design*.

[Pastre, 89] *MUSCADET: An automatic theorem proving system using knowledge and metaknowledge in mathematics*. AI J. 38, 3, pp 257-318.

[Pitrat 66] *A theorem-proving program using heuristic methods*. PhD dissertation, Paris 6 University, 1966.

[Smith, 91] *KIDS: A semiautomatic program development system*. In Lowry & McCartney, Eds, *Automating software design*.

[Steier, Anderson, 89] *Algorithm synthesis : A comparative study*. Springer-Verlag. New-York.