Electronic Notes in Theoretical Computer Science 70 No. 2 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume70.html` 13 pages

Producing Proofs from an Arithmetic Decision Procedure in Elliptical LF

Aaron Stump, Clark W. Barrett, and David L. Dill

Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA
E-mail: {stump,barrett,dill}@cs.stanford.edu
Phone: +1 650 725 3646, Fax: +1 650 725 6949

Abstract

Software that can produce independently checkable evidence for the correctness of its output has received recent attention for use in certifying compilers and proof-carrying code. CVC ("a Cooperating Validity Checker) is a proof-producing validity checker for a decidable fragment of first-order logic enriched with background theories. This paper describes how proofs of valid formulas are produced from the decision procedure for linear real arithmetic implemented in CVC. It is shown how extensions to LF which support proof rules schematic in an arity ("elliptical" rules) are very convenient for this purpose.

1 Introduction

The ability for automated reasoning systems to produce easily verifiable proofs has been widely recognized as valuable (see, e.g., [14,4]). Recently, applications like proof-carrying code and proof-carrying authentication have created a new need for proofs [9,1]. The Edinburgh Logical Framework (LF) [7] has emerged as a widely used meta-language for representing proof systems for these applications. The representation is such that proof checking is reduced to LF type checking.

CVC ("a Cooperating Validity Checker") is a system based on the Nelson-Oppen method for combining decision procedures (DPs) for certain quantifier-free first-order classical theories [13]. CVC produces proofs for formulas it claims are valid. Each subsidiary DP produces proofs for all the reasoning it performs. The proofs are represented in an extended version of LF, called $\mathrm{LF}_{\mathrm{nat}}$. In $\mathrm{LF}_{\mathrm{nat}}$, lists can be represented in a direct way. The goal of this paper is to show how the features of $\mathrm{LF}_{\mathrm{nat}}$ make it convenient to produce proofs from a fairly complex decision procedure, the DP for linear real arithmetic.

The organization is as follows. Section 2 presents selected parts of the DP for the theory of linear real arithmetic. Section 3 discusses some difficulties that arise using pure LF for representing proofs. Section 4 describes $\mathrm{LF}_{\mathrm{nat}}$, and Section 5 shows how proof rules corresponding to the steps of reasoning in the decision procedure are represented in $\mathrm{LF}_{\mathrm{nat}}$. Some familiarity with LF or other type theories with dependent types must be assumed (see, e.g., [10]). This paper describes work in progress.

2 Deciding linear real arithmetic in CVC

In linear real arithmetic, terms are built using binary function symbols $+$, $*$, and $/$; a unary function symbol $-$; numerals 0, 1, 2, etc.; and variables from a countably infinite set $\mathcal{V}$. Variables range over the reals. The restriction of linearity is that no term may contain a subterm of the form $t * t'$, where t and t' both contain a variable. Also, we restrict divisors to non-zero numerals. Atomic formulas are then equations $t = t'$ and inequalities $t < t'$ between terms. As usual, literals are atomic formulas or their negations.

In CVC, propositional reasoning is separated from theory-specific reasoning (see [2]). To describe a DP for a theory, it is sufficient to present an algorithm for testing a set of literals for satisfiability. In CVC, such a set of literals is presented to the DP one literal at a time. The DP is responsible for notifying the rest of the system whenever the set of literals it has been given so far is unsatisfiable. Giving a literal to a DP is called *asserting* the literal to the DP.

It suffices then to show how the DP for linear real arithmetic handles each literal that may be asserted to it. The DP consists of three main components: a canonizer for arithmetic terms, a solver for asserted equations, and a component that implements Fourier-Motzkin variable elimination to handle asserted inequalities. We will now describe the canonizer and solver in detail. The section about the solver explains how disequalities $t \neq t'$ are handled. We will not describe how inequalities and negated inequalities are handled, for reasons of space.

2.1 Canonizer for terms

It is convenient to put terms into a canonical form for use by the solver and the Fourier-Motzkin component. This section describes how this canonization of terms is performed.

Let $\prec$ be an arbitrary fixed total ordering of the set $\mathcal{V}$ of variables. A constant arithmetic term is in *coprime* form iff it is of the form N/D, where N and D are relatively prime numerals, and D is positive. A natural canonical form for a linear arithmetic term is as a polynomial $c + c_1 * x_1 + \ldots c_n * x_n$, where

(i) $c, c_1, \ldots, c_n$ are constant arithmetic terms in coprime form

(ii) $x_1, \ldots, x_n \in \mathcal{V}$

(iii) for all $1 \leq i < j \leq n$, $x_i \prec x_j$

We consider now how to implement an algorithm in the CVC system to put linear arithmetic terms into such a form.

The natural and convenient internal implementation of polynomials in CVC is as applications of an n-ary addition operator to monomials: e.g., $3*x+4*y+z$ is represented as $(+\ (3*x)\ (4*y)\ z)$. This is a good representation because in the CVC infrastructure, any immediate child of an expression may be accessed in constant time. In constrast, if polynomials were represented as nested applications of a binary addition operator, accessing the i'th monomial of a polynomial would take $O(i)$ time. So it is more efficient to represent polynomials in this flattened form as applications of an n-ary addition operator. The first step in canonizing a linear arithmetic term is to put it into flattened form.

2.1.1 Reducing to flattened form

This section shows how a linear arithmetic term is put into the flattened form $(+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n))$, where $c, c_1, \ldots, c_n$ are constant arithmetic terms and $x_1, \ldots, x_n \in \mathcal{V}$, but like terms ($c_i * x_i$ and $c_j * x_j$ with $x_i \equiv x_j$) are not yet combined nor monomials sorted according to the ordering $\prec$ on variables. The latter operations happen in the next step of canonization. Several special cases of flattened form are for constant terms c, where the flattened form $(+\ c)$ has n equal to 0; and for variables, where the flattened form is $(+\ 0\ (1 * v))$. It simplifies the statement of the flattening algorithm to put constants and variables temporarily into these forms.

The main rewriting engine in CVC is a simplifier that rewrites literals bottom-up, before they are asserted to subsidiary DPs. Theory-specific rewrites may be registered for that theory's function and predicate symbols. Those rewrites will be called by the simplifier to try to rewrite subterms after the subterms' children have been fully simplified. The algorithm to put a term in flattened form is implemented in a theory-specific rewrite registered for the arithmetic function symbols ($+$, $*$, $/$, and $-$). It consists in the rewrites given in Figure 2.1.1, which assume that the immediate subexpressions of the term being rewritten are already in flattened form. There are two rules for plus: the first is for when there are two summands, and the second is for when there are more than two (assume $m > 0$ in that rule). The rules for multiplication and division assume that the left multiplicand and the divisor, respectively, are constant arithmetic terms. We omit the similar rule for multiplication where the right multiplicand is a constant term.

2.1.2 Completing the canonization

Once a term is in flattened form, canonization may be completed by sorting the monomials so that $c * x$ comes before $d * y$ if $x \prec y$. In the resulting polynomial, all like terms $c_1 * x, \ldots, c_n * x$ with the same variable will appear

[uminus] $-(+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n)) \quad \longrightarrow$
$(+\ (-c)\ (-c_1 * x_1)\ \ldots\ (-c_n * x_n))$

[mult] $c' * (+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n)) \quad \longrightarrow$
$(+\ (c' * c)\ ((c' * c_1) * x_1)\ \ldots\ ((c' * c_n) * x_n))$

[division] $(+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n))\,/\,c' \quad \longrightarrow$
$(+\ (c/c')\ ((c_1/c') * x_1)\ \ldots\ ((c_n/c') * x_n))$

[addition] $(+\ (+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n))\ (+\ d\ (d_1 * y_1)\ \ldots\ (d_{n'} * y_{n'})))$
$\longrightarrow \quad (+\ (c+d)\ (c_1 * x_1)\ \ldots\ (c_n * x_n)\ (d_1 * y_1)\ \ldots\ (d_{n'} * y_{n'}))$

[addition] $(+\ t_1\ \ldots\ t_m$
$(+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n))\ (+\ d\ (d_1 * y_1)\ \ldots\ (d_{n'} * y_{n'})))$
$\longrightarrow \quad (+\ t_1\ \ldots\ t_m$
$(+\ (c+d)\ (c_1 * x_1)\ \ldots\ (c_n * x_n)\ (d_1 * y_1)\ \ldots\ (d_{n'} * y_{n'})))$

[constant] $c \quad \longrightarrow \quad (+\ c)$

[variable] $v \quad \longrightarrow \quad (+\ 0\ (1 * v))$

Fig. 1. Rewrites to flatten a linear arithmetic term

next to each other. Those terms may be combined by replacing them in the polynomial with $(c_1 + \ldots c_n) * x$. To complete the canonization, constant arithmetic terms like $2/4 + 3/6$ are put into coprime form. Finally, the trivial cases of flattened forms are rewritten:

$$(+\ (1 * v)\ 0) \quad \longrightarrow \quad v \qquad\qquad (+\ c) \quad \longrightarrow \quad c$$

2.2 Solver

We consider now how to handle an asserted equation $t = t'$, where t and t' are in canonical form.

2.2.1 Canonizing equations

The equation is first put into a canonical form $\hat{t} = 0$, where $\hat{t}$ is the canonical form of the term $t + (-t')$. If $\hat{t}$ is a (canonical) constant arithmetic term, then if $\hat{t} \equiv 0$ this equation is equivalent to *TRUE* and can be dropped; and otherwise, it is equivalent to *FALSE*, and the DP reports that the set of asserted literals is unsatisfiable. If $\hat{t}$ is not a constant term, then solving proceeds on $\hat{t} = 0$ by isolating a variable.

2.2.2 Isolating a variable

The equation at this point is in the form

$$(+\ c\ (c_1 * x_1)\ \ldots\ (c_n * x_n)) = 0$$

A variable is chosen from the left hand side of the equation to isolate. Currently, the variable x_i with the shortest *notify list* (a list of tasks to perform when an equation with x_i as its left hand side is asserted) is chosen. Then the equation is rewritten to

$$x_i = (+\ d\ (d_1 * x_1)\ \ldots\ (d_{i-1} * x_{i-1})\ (d_{i+1} * x_{i+1})\ \ldots\ (d_n * x_n)),$$

where for all $j \in \{1, \ldots, i-1, i+1, \ldots n\}$, d_j is the coprime form of $(-c_j/c_i)$; and d is the coprime form of $(-c/c_i)$. This equation is then committed to a database of equalities, maintained by the infrastructure of CVC. This will cause x_i to be replaced by the right hand side of the equation in all literals in the arithmetic theory. This eliminates the variable x_i from those literals. Literals asserted subsequently will have this replacement applied to them as well.

2.2.3 Handling disequalities

Disequalities like $t \neq t'$ are handled by simply replacing x with y in the disequality (if the disequality contains x) whenever an equation $x = y$ is committed to the database of equalities in CVC. Any time the disequality is updated in this way and also when it is first asserted, the canonizer for equations (described just above) is run on $t = t'$, to make sure that it does not canonize to *TRUE*. If it does canonize to *TRUE*, then the DP reports that the set of literals asserted so far is unsatisfiable.

3 Producing proofs

Now that the canonizer and solver from the DP for linear real arithmetic have been described, we turn to the problem of producing proofs. The basic approach we follow is to instrument the code for the DP so that every time a step of reasoning is performed, the appropriate inference is added to the proof. CVC's infrastructure is already instrumented to produce proofs. For example, the simplifier glues together the proofs produced for single steps of theory-specific rewriting. The proofs are glued together using proof rules for transitivity and congruence of equality. Also, the database of equalities expects and stores a proof for each equality that is committed to it.

The more challenging part of the problem of proof production is how to describe the inference rules for the decision procedure in LF. In our case, if we use pure LF, difficulties arise right away with the representation of polynomials. As discussed above (Section 2.1), the DP works with polynomials that are applications of an n-ary addition operator. LF does not provide direct support for arity-polymorphic operators like this, so a polynomial must be represented less directly in LF. It could be encoded as either nested applications of a binary addition operator (e.g., $(+\ 5\ (+\ (3*x)\ (4*y))))$ or as an application of an addition operator to a list of monomials (e.g., $(+\ (\mathit{cons}\ 5\ (\mathit{cons}\ (3*x)\ (\mathit{cons}\ (4*y)\ \mathit{null})))))$. Both representations are indirect, in the sense that a single function application (of n-ary addition) in the object language is represented by an asymptotically greater number (n) of function applications in LF.

With indirect representations of polynomials, simple inferences in the DP also require an indirect representation. For example, with an indirect representation of polynomials, isolating the i'th monomial of a polynomial first requires proving that isolated monomial is indeed the i'th monomial. Consider, for example, the isolation

$$(+\ 5\ (3*x)\ (4*y)) = 0 \quad \longrightarrow \quad -4*y = (+\ 5\ (3*x)).$$

If the first equation is represented as something like $(\mathit{eq}\ (+\ (\mathit{cons}\ 5\ (\mathit{cons}\ (3*x)\ (\mathit{cons}\ (4*y)\ \mathit{null})))\ 0)$, then the most natural LF encoding for the isolation of the i'th monomial will require something like the following rule (we use the concrete LF syntax of Twelf [11], where $\lambda\, x : A.M$ is written $[x : A]M$ and $\Pi\, x : A.B$ is written $\{x : A\}B$):

```
isolate_eq : { L : list } { L' : list }
             { t : trm real } { i : natural }
             { P : (remove_ith i L t L') }
             (pf (eq (eq (+ L) 0) (eq (minus t) (+ L'))))
```

This rule says, assuming suitable other declarations, that isolating the i'th monomial t in an equation $(+\ L) = 0$ yields an equation $-t = (+L')$, as long as it can be proved that removing the i'th monomial from L yields the term t, as well as the version L' of L that has t removed. The problem with this rule is that it requires a proof of $(\mathit{remove\_ith}\ i\ L\ t\ L')$, which can be proved most naturally only with a proof of size $O(i)$. It is possible that less natural representations would yield a proof of size $O(1)$, but coming up with such a representation places an additional burden on the proof producer. If the natural approach is used, additional C++ code will be required to generate the proof of $(\mathit{remove\_ith}\ i\ L\ t\ L')$. While this additional code may not be a very heavy burden, it is desirable to try to minimize the amount of extra code required to produce proofs from the decision procedure.

Frank Pfenning pointed out at the LFM '02 workshop that lists (in this case, of monomials) could be appended without encoding proof rules for append, if the lists were represented as functions taking in the rest of the members of the list. So a list $[a, b, c]$ could be represented as

$$\lambda\, r : \mathit{list}.(\mathit{cons}\ a\ (\mathit{cons}\ b\ (\mathit{cons}\ c\ r)))$$

A list L_1 of type $\mathit{list} \rightarrow \mathit{list}$ can then be appended to list L_2 of the same type by forming $\lambda\, r : \mathit{list}.(L_1\ (L_2\ r))$. This technique does not readily make it possible, however, to extract the i'th element of a list, as is needed for isolating a monomial of an equation.

4 Elliptical LF

We consider now an extension of LF called $\mathrm{LF}_{\mathrm{nat}}$, which helps make the natural representation of proofs from the arithmetic DP more convenient. $\mathrm{LF}_{\mathrm{nat}}$ is supported by a tool called **flea**, which ships with CVC. $\mathrm{LF}_{\mathrm{nat}}$ adds to LF a basic type *nat* of natural numbers and numerals of type *nat*. There are also subrange types $[l..u]$ which are subtypes of *nat*. This makes it possible to model a list of n elements as a function with domain $[1..n]$. The basic arithmetic operations of addition and subtraction are convenient to add, as well as arithmetic inequality. We can add multiplication and division if we allow arithmetic operators to operate on a supertype *real* of *nat*. We can then either require our subrange type constructor to accept *real*s (and perhaps represent the set of *nat*s in that range), or else enforce that subrange types can only be built from *nat*s. As mentioned just below, $\mathrm{LF}_{\mathrm{nat}}$ is work in progress, and a number of issues like these have yet to be completely resolved.

$\mathrm{LF}_{\mathrm{nat}}$ also adds an *if-then-else* construct, so that an actual list like $[a, b, c]$ can be modeled as

$$\lambda\, i : \mathit{nat}.\ \mathit{if}\ (i = 1)\ \mathit{then}\ a\ \mathit{else}\ \mathit{if}\ (i = 2)\ \mathit{then}\ b\ \mathit{else}\ c\ \mathit{end}\ \mathit{end}$$

The concrete syntax of flea allows this λ-abstraction to be denoted $[[a, b, c]]$. The guarding formula of an *if-then-else* is required to be either an equation or an inequality between terms of type *nat*. It is also convenient to allow polytypic lists. If $a : A$, $b : B$, and $c : C$, we might like the list $[[a, b, c]]$ to have type corresponding to something like $[A, B, C]$. We can achieve this by making the type of an *if-then-else* expression also be an *if-then-else* expression:

$$\mathit{if}\ \phi\ \mathit{then}\ M\ \mathit{else}\ M'\ \mathit{end} \quad : \quad \mathit{if}\ \phi\ \mathit{then}\ A\ \mathit{else}\ A'\ \mathit{end},$$

where ϕ implies $M : A$ and $\neg\phi$ implies $M' : A'$. Then the type of $[[a, b, c]]$ is:

$$\Pi\, i : \mathit{nat}.\ \mathit{if}\ (i = 1)\ \mathit{then}\ A\ \mathit{else}\ \mathit{if}\ (i = 2)\ \mathit{then}\ B\ \mathit{else}\ C\ \mathit{end}\ \mathit{end}.$$

The concrete syntax of flea allows this Π-abstraction to be denoted $\{\{A, B, C\}\}$.

4.1 Metatheory

The study of $\mathrm{LF}_{\mathrm{nat}}$ is work in progress. Just giving a formal description of the system is quite complex. The system of Dependent Predicate Logic

(DPL) of [8], which adds predicate logic on top of a dependent type theory, could be a starting point. Difficulties may arise, because it appears that in DPL, the form of sequents is such that a classification $X : Y$ is not allowed to depend on a logical assumption. This may make it difficult to formulate suitable rules for *if-then-else*. A more fundamental issue that remains unclear is the treatment of the partiality of functions which are declared to have a type that is a proper subtype; in particular, how to make the treatment of partiality follow that proposed for bivalent logics of partial terms [3,6,5].

One thing that is known is that just adding the type *nat* and subrange types to LF makes type-checking undecidable in general. This result can be proved by reducing the word problem to typability in $\mathrm{LF}_{\mathrm{nat}}$. The crucial idea is that by declaring certain types to be inhabited, we can add a logical assumption of the form $\forall\, x : \mathit{nat}.\, t = t'$, where t and t' are built from x and function symbols of type $\mathit{nat} \rightarrow \mathit{nat}$. We just add the typing declarations:

$$g_1 : \Pi\, x : \mathit{nat}.\, [t..t'].$$

$$g_2 : \Pi\, x : \mathit{nat}.\, [t'..t].$$

The only way $\Pi\, x : \mathit{nat}.\, [t..t']$ and $\Pi\, x : \mathit{nat}.\, [t'..t]$ can both be inhabited is if $\forall\, x : \mathit{nat}.\, t = t'$. Adding a logical assumption of this form is the crucial step in reducing the word problem to typability.

4.2 Consequences for checking proofs from CVC

Since $\mathrm{LF}_{\mathrm{nat}}$ is undecidable, it might seem hopeless to use it as a logical framework for checking proofs from CVC. But undecidability is not as serious a problem as one might expect in this particular case, for the following reason. Suppose the only objects (including bound variables) of type *nat* in some expression E are numerals or of type $[n..m]$, for numerals n and m. Suppose that the types of all declared function symbols are such that no term of type *nat* may be formed by application from those function symbols. Type-checking can then be decided using only evaluation of constant arithmetic expressions and possibly trying all values of finite types like $[3..5]$. The latter is used when a bound variable declared to have such a finite type is added to the context by the usual (lam) rule for classifying lambda abstractons.

The typing declarations corresponding to the proof rules for CVC will not always satisfy these conditions which make type-checking in $\mathrm{LF}_{\mathrm{nat}}$ decidable. But the proof system is designed so that every actual proof object produced by CVC for a valid formula will. This means that while it may not be possible to check the types of the encodings of CVC's proof rules, it will always be possible to check the type for an actual proof. This is good enough for CVC's purposes.

5 Representing arithmetic proofs in $\mathrm{LF}_{\mathrm{nat}}$

This section presents the encoding of the proof rules used by the arithmetic DP described in Section 2. The syntax used is that of flea, which extends the syntax for Twelf in the ways mentioned in the previous section. The focus is on representing each step of reasoning performed by the DP in as direct a way as possible, using the features of $\mathrm{LF}_{\mathrm{nat}}$. Deriving the rules from a simpler set of axioms is not considered. We begin with a few standard declarations, which are used in the representation of arithmetic.

```
sort : type.
trm : sort -> type.
BOOLEAN : sort.
o = (trm BOOLEAN).
EQ : {S:sort} (trm S) -> (trm S) -> o.
TRUE : o.
FALSE : o.
```

As usual, proofs of a theorem F will be represented as objects of type $(pf\ F)$:

```
pf : o -> type.
```

Now we turn to the representation of the reasoning for the arithmetic DP. We first represent the terms and atomic formulas of the language. Inequality is omitted, because we consider only the rules for the canonizer and solver. PLUS is declared to take in a natural number n and a list of n REAL terms, where the list is modeled as a function from $[1..n]$ to $(trm\ REAL)$:

```
REAL : sort.
PLUS : {n:nat} ([1..n] -> trm REAL) -> trm REAL.
MULT : trm REAL -> trm REAL -> trm REAL.
DIVIDE : trm REAL -> trm REAL -> trm REAL.
UMINUS : trm REAL -> trm REAL.
```

We embed *real*s as REAL terms, and we add an abbreviation for polynomials

$$(+\ c\ (c_1 * t_1)\ \ldots\ (c_n * t_n)),$$

where $c, c_1, \ldots, c_n$ are (embeddings) of *real*s, and $t_1, \ldots, t_n$ are REAL terms. For the benefit of the actual proof-production code in CVC, it is convenient if we can use this abbreviation with a list of ordered pairs (c_i, t_i). The type $\{\{real,\ (trm\ REAL)\}\}$ is the type used for these pairs (for the "$\{\{\ \}\}$" notation, see Section 4 just above).

```
n2I : nat -> (trm REAL).
CPLUS =
  [n:nat]  [C : nat]
  [c : [1..n] -> {{real, (trm REAL)}} ]
  (PLUS (n+1) [i:[1..n+1] ]
              if i == 1 then (n2I C)
              else MULT (n2I (c (i - 1) 1)) (c (i - 1) 2)
              end).
```

So the representation of $(+\ 5\ (5 * x)\ (5 * y))$ can be written as

```
(CPLUS 2 5 [[ [[5,x]], [[5,y]] ]]),
```

which normalizes to:

```
(PLUS 3 [[(n2I 5), (MULT (n2I 5) x), (MULT (n2I 5) y)]]).
```

The rewrite rule from Figure 2.1.1 for flattening an application of unary minus is represented as follows ($[x : A]M$ is Twelf syntax for $\lambda\, x : A.M$):

```
arith_canon_uminus :
  {n:nat}  {C : nat}  {c : [1..n] -> {{real, (trm REAL)}} }
  (pf (EQ REAL
        (UMINUS (CPLUS n C c))
        (CPLUS n (-C)
           [i:[1..n] ] [[(- (c i 1)), (c i 2)]]))).
```

Notice that the built-in unary minus operation of $\mathrm{LF}_{\mathrm{nat}}$ is used. The effect is that polynomials are represented modulo constant arithmetic. So constant arithmetic terms with the same value have equal representations in $\mathrm{LF}_{\mathrm{nat}}$. An example application of this rule is:

```
(arith_canon_uminus 2 4 [[ [[1,c]] [[-1,d]] ]]),
```

which has type

```
(pf (EQ REAL
       (UMINUS (PLUS 3 [[ (n2I 4), (MULT (n2I 1) c),
                           (MULT (n2I -1) d) ]]))
       (PLUS 3 [[ (n2I -4), (MULT (n2I -1) c),
                  (MULT (n2I 1) d) ]])))
```

One inference performed by the solver is: given an equation $t = 0$, where t is a polynomial in canonical form whose j'th monomial (not counting the initial constant in the polynomial) is $D * x$, isolate x on the left hand side of the equation. This inference is represented as follows:

```
isolate_var :
  {n : nat}  {C : nat}  {c : [1..n] -> {{real, trm REAL}} }
  {j : [1..n]}  {D : real}  {x : trm REAL}
  (pf (EQ BOOLEAN
         (EQ REAL (CPLUS n C
                    [i:[1..n] ] if i == j then [[ D,  x]]
                                            else (c i) end)
                   (n2I 0))
         (EQ REAL x
           (UMINUS (DIVIDE (CPLUS (n - 1) C
                              [i: [1..(n - 1)] ]
                                if i < j then (c i)
                                else (c (i + 1)) end)
                            (n2I D)))))).
```

Notice how *if-then-else* is used first to say that the j'th non-constant monomial is $D * x$, and then to say that in the equation with x isolated, the sum in the right hand side skips over that j'th term. Notice also that the rule cannot be fully applied when n equals 0, because there will be no j of type [1..0], since that type is empty. This is assuming that the typing context is consistent, in the sense that it does not declare empty types to be inhabited.

As a final example, we include a representation of the inference in the canonizer which sorts the terms of a polynomial. It is quite undesirable from an engineering perspective to instrument a sorting routine to produce a proof. Without any modification to the sorting routine, however, we can use an auxiliary data structure to recover the permutation relating the unsorted and sorted lists. We can then give a rule which says that applying a permutation p to a list c of summands yields an equal sum. In addition to the permutation p, the rule takes in p's inverse, *ip*. It then requires a proof *dp* that p composed with *ip* is the identity. This guarantees that p is really a permutation (and that *ip* really is its inverse). The rule says that *dp* must be of type *composition* == *identity*; flea allows == to be used as a type. flea then just checks that the two expressions are actually equal according to the definitional equality of $\mathrm{LF}_{\mathrm{nat}}$. flea requires a placeholder argument "**" to be supplied as the argument corresponding to the == type.

```
sort_summands :
  {n:nat}  {c : [1..n] -> trm REAL}
  {p : [1..n] -> [1..n]}
  {ip : [1..n] -> [1..n]}
  {dp : (([i:[1..n] ] (ip (p i))) == ([i:[1..n] ] i)) }
  (pf (EQ REAL
        (PLUS n c)
        (PLUS n [k:[1..n] ] c (p k)))).
```

An example application of this rule (assuming a, b, and c of type (*trm REAL*)) is

```
(sort_summands 3
   [[ a, b, c ]] [[ 3, 1, 2 ]] [[ 2, 3, 1 ]] **),
```

which is of type

```
(pf (EQ REAL
      (PLUS 3 [[ a, b, c ]] )
      (PLUS 3 [[ c, a, b]]))).
```

6 Conclusion

The canonizer and solver from CVC's proof-producing decision procedure for linear real arithmetic have been described. The proofs produced are represented in $\mathrm{LF}_{\mathrm{nat}}$, an extension of LF with numerals, subrange types, arithmetic operations, and *if-then-else*. $\mathrm{LF}_{\mathrm{nat}}$ makes it more convenient to model the reasoning in this and other decision procedures in CVC than in pure LF. Much work remains if a rigorous understanding of the metatheory of $\mathrm{LF}_{\mathrm{nat}}$ is to be obtained. Even though $\mathrm{LF}_{\mathrm{nat}}$ is undecidable in general, it can still correctly check the proofs produced by CVC. Arity-polymorphic proof rules may fail to be checked by flea, but all actual proofs of formulas CVC claims are valid can be checked. It may be that the proof rules used by CVC fall into a decidable fragment of $\mathrm{LF}_{\mathrm{nat}}$, but this remains to be seen.

This work was supported under ARPA/Air Force contract F33615-00-C-1693 and NSF contract CCR-9806889.

References

[1] A. Appel and E. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communication Security*, 1999.

[2] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.

[3] M. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.

[4] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic . In *Theorem Proving in Higher Order Logics, 13th International Conference*, volume 1869 of *LNCS*, 2000.

[5] W. Farmer and J. Guttman. A Set Theory with Support for Partial Functions. *Logica Studia*, 66(1):59–78, 2000.

[6] S. Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995.

[7] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[8] B. Jacobs. *Categorical Logic and Type Theory.* Elsevier, 1999.

[9] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[10] F. Pfenning. *Logical Frameworks*, chapter XXI. In Robinson and Voronkov [12], 2001.

[11] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[12] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier and MIT Press, 2001.

[13] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[14] W. Wong. Validation of HOL Proofs by Proof Checking. *Formal Methods in System Design*, 14(2):193–212, 1999.