Formalizing the Logic-Automaton Connection

Stefan Berghofer* and Markus Reiter

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany

Abstract. This paper presents a formalization of a library for automata on bit strings in the theorem prover Isabelle/HOL. It forms the basis of a reflection-based decision procedure for Presburger arithmetic, which is efficiently executable thanks to Isabelle's code generator. With this work, we therefore provide a mechanized proof of the well-known connection between logic and automata theory.

1 Introduction

Although higher-order logic (HOL) is undecidable in general, there are many decidable logics such as *Presburger arithmetic* or the *Weak Second-order theory of One Successor* (WS1S) that can be embedded into HOL. Since HOL can be viewed as a logic containing a functional programming language, an interesting approach for implementing a decision procedure for such a decidable logic in a theorem prover based on HOL is to write *and verify* the decision procedure as a recursive function in HOL itself. This approach, which is called *reflection* [7], has been used in proof assistants based on type theory for quite a long time. For example, Boutin [4] has used reflection to implement a decision procedure for abelian rings in Coq. Recently, reflection has also gained considerable attention in the Isabelle/HOL community. Chaieb and Nipkow have used this technique to verify various quantifier elimination procedures for dense linear orders, real and integer linear arithmetic, as well as Presburger arithmetic [5, 12]. While the decision procedures by Chaieb and Nipkow are based on *algebraic* methods like Cooper's algorithm, there are also *semantic* methods, as implemented e.g. in the MONA tool [8] for deciding WS1S formulae. In order to check the validity of a formula, MONA translates it to an automaton on bitstrings and then checks whether it has accepting states. Basin and Friedrich [1] have connected MONA to Isabelle/HOL using an *oracle-based* approach, i.e. they simply trust the answer of the tool. As a motivation for their design decision, they write:

> Hooking an 'oracle' to a theorem prover is risky business. The oracle could be buggy [...]. The only way to avoid a buggy oracle is to reconstruct a proof in the theorem prover based on output from the oracle, or perhaps verify the oracle itself. For a semantics based decision procedure, proof reconstruction is not a realistic option: one would have to formalize the entire automata-theoretic machinery within HOL [...].

* Supported by BMBF in the VerisoftXT project under grant 01 IS 07008 F

In this paper, we show that verifying decision procedures based on automata in HOL is not as unrealistic as it may seem. We develop a library for automata on bitstrings, including operations like forming the product of two automata, projection, and determinization of nondeterministic automata, which we then use to build a decision procedure for Presburger arithmetic. The procedure can easily be changed to cover WS1S, by just exchanging the automata for atomic formulae. The specification of the decision procedure is completely executable, and efficient ML code can be generated from it using Isabelle's code generator [2]. To the best of our knowledge, this is the first formalization of an automata-based decision procedure for Presburger arithmetic in a theorem prover.

The paper is structured as follows. In §2, we introduce basic concepts such as Presburger arithmetic, automata theory, bit vectors, and BDDs. The library for automata is described in §3. The actual decision procedure together with its correctness proof is presented in §4, and §5 draws some conclusions. Due to lack of space, we will not discuss any of the proofs in detail. However, the interested reader can find the complete formalization on the web[1].

2 Basic Definitions

2.1 Presburger Arithmetic

Formulae of Presburger arithmetic are represented by the following datatype:

datatype $pf = Eq\ (int\ list)\ int \mid Le\ (int\ list)\ int \mid And\ pf\ pf \mid Or\ pf\ pf$
$\mid Imp\ pf\ pf \mid Forall\ pf \mid Exist\ pf \mid Neg\ pf$

The atomic formulae are *Diophantine (in)equations* $Eq\ ks\ l$ and $Le\ ks\ l$, where ks are the (integer) coefficients and l is the right-hand side. Variables are encoded using de Bruijn indices, meaning that the ith coefficient in ks belongs to the variable with index i. Thus, the well-known *stamp problem*

$$\forall\, x{\geq}8.\ \exists\, y\ z.\ 3 * y + 5 * z = x$$

can be encoded by

$$Forall\ (Imp\ (Le\ [-1]\ -8)\ (Exist\ (Exist\ (Eq\ [5,\ 3,\ -1]\ 0))))$$

Like Boudet and Comon [3], we only consider variables ranging over the natural numbers. The left-hand side of a Diophantine (in)equation with variables xs can be evaluated using the function

$$\begin{array}{l} eval\text{-}dioph :: int\ list \Rightarrow nat\ list \Rightarrow int \\ eval\text{-}dioph\ (k \cdot ks)\ (x \cdot xs) = k * int\ x + eval\text{-}dioph\ ks\ xs \\ eval\text{-}dioph\ []\ xs = 0 \end{array}$$

[1] `http://www.in.tum.de/~berghofe/papers/automata`

where *int* coerces a natural number to an integer, and $x \cdot xs$ denotes the '*Cons*' operator. Given a valuation *xs*, a Presburger formula can be evaluated by

$$\begin{array}{l}
\mathit{eval\text{-}pf} :: \mathit{pf} \Rightarrow \mathit{nat\ list} \Rightarrow \mathit{bool} \\
\mathit{eval\text{-}pf}\ (\mathit{Eq}\ \mathit{ks}\ l)\ \mathit{xs} = (\mathit{eval\text{-}dioph}\ \mathit{ks}\ \mathit{xs} = l) \\
\mathit{eval\text{-}pf}\ (\mathit{Le}\ \mathit{ks}\ l)\ \mathit{xs} = (\mathit{eval\text{-}dioph}\ \mathit{ks}\ \mathit{xs} \leq l) \\
\mathit{eval\text{-}pf}\ (\mathit{Neg}\ p)\ \mathit{xs} = (\neg\ \mathit{eval\text{-}pf}\ p\ \mathit{xs}) \\
\mathit{eval\text{-}pf}\ (\mathit{And}\ p\ q)\ \mathit{xs} = (\mathit{eval\text{-}pf}\ p\ \mathit{xs} \wedge \mathit{eval\text{-}pf}\ q\ \mathit{xs}) \\
\mathit{eval\text{-}pf}\ (\mathit{Or}\ p\ q)\ \mathit{xs} = (\mathit{eval\text{-}pf}\ p\ \mathit{xs} \vee \mathit{eval\text{-}pf}\ q\ \mathit{xs}) \\
\mathit{eval\text{-}pf}\ (\mathit{Imp}\ p\ q)\ \mathit{xs} = (\mathit{eval\text{-}pf}\ p\ \mathit{xs} \longrightarrow \mathit{eval\text{-}pf}\ q\ \mathit{xs}) \\
\mathit{eval\text{-}pf}\ (\mathit{Forall}\ p)\ \mathit{xs} = (\forall x.\ \mathit{eval\text{-}pf}\ p\ (x \cdot \mathit{xs})) \\
\mathit{eval\text{-}pf}\ (\mathit{Exist}\ p)\ \mathit{xs} = (\exists x.\ \mathit{eval\text{-}pf}\ p\ (x \cdot \mathit{xs}))
\end{array}$$

2.2 Abstract Automata

The abstract framework for automata used in this paper is quite similar to the one used by Nipkow [11]. The purpose of this framework is to factor out all properties that deterministic and nondeterministic automata have in common. Automata are characterized by a transition function *tr* of type $\sigma \Rightarrow \alpha \Rightarrow \sigma$, where σ and α denote the types of *states* and *input symbols*, respectively. Transition functions can be extended to *words*, i.e. lists of symbols in a canonical way:

$$\begin{array}{l}
\mathit{steps} :: (\sigma \Rightarrow \alpha \Rightarrow \sigma) \Rightarrow \sigma \Rightarrow \alpha\ \mathit{list} \Rightarrow \sigma \\
\mathit{steps}\ \mathit{tr}\ q\ [] = q \\
\mathit{steps}\ \mathit{tr}\ q\ (a \cdot \mathit{as}) = \mathit{steps}\ \mathit{tr}\ (\mathit{tr}\ q\ a)\ \mathit{as}
\end{array}$$

The *reachability* of a state *q* from a state *p* via a word *as* is defined by

$$\begin{array}{l}
\mathit{reach} :: (\sigma \Rightarrow \alpha \Rightarrow \sigma) \Rightarrow \sigma \Rightarrow \alpha\ \mathit{list} \Rightarrow \sigma \Rightarrow \mathit{bool} \\
\mathit{reach}\ \mathit{tr}\ p\ \mathit{as}\ q \equiv q = \mathit{steps}\ \mathit{tr}\ p\ \mathit{as}
\end{array}$$

Another characteristic property of an automaton is its set of accepting states. Given a predicate *P* denoting the accepting states, an automaton is said to accept a word *as* iff from a starting state *s* we reach an accepting state via *as*:

$$\begin{array}{l}
\mathit{accepts} :: (\sigma \Rightarrow \alpha \Rightarrow \sigma) \Rightarrow (\sigma \Rightarrow \mathit{bool}) \Rightarrow \sigma \Rightarrow \alpha\ \mathit{list} \Rightarrow \mathit{bool} \\
\mathit{accepts}\ \mathit{tr}\ P\ s\ \mathit{as} \equiv P\ (\mathit{steps}\ \mathit{tr}\ s\ \mathit{as})
\end{array}$$

2.3 Bit Vectors and BDDs

The automata used in the formalization of our decision procedure for Presburger arithmetic are of a specific kind: the input symbols of an automaton corresponding to a formula with *n* free variables $x_0, \ldots, x_{n-1}$ are bit lists of length *n*.

$$\begin{array}{c} x_0 \\ \vdots \\ x_{n-1} \end{array} \left[\begin{bmatrix} b_{0,0} \\ \vdots \\ b_{n-1,0} \end{bmatrix} \begin{bmatrix} b_{0,1} \\ \vdots \\ b_{n-1,1} \end{bmatrix} \begin{bmatrix} b_{0,2} \\ \vdots \\ b_{n-1,2} \end{bmatrix} \cdots \begin{bmatrix} b_{0,m-1} \\ \vdots \\ b_{n-1,m-1} \end{bmatrix} \right]$$

The rows in the above word are interpreted as *natural numbers*, where the left-most column, i.e. the first symbol in the list corresponds to the *least significant bit*. Therefore, the value of variable x_i is $\sum_{j=0}^{m-1} b_{i,j} 2^j$. The list of values of n variables denoted by a word can be computed recursively as follows:

nats-of-boolss :: *nat* ⇒ *bool list list* ⇒ *nat list*
nats-of-boolss n [] = *replicate n 0*
nats-of-boolss n (*bs* · *bss*) =
  map (λ(*b*, *x*). *nat-of-bool b* + *2* ∗ *x*) (*zip bs* (*nats-of-boolss n bss*))

where *zip* [b_0, b_1, ...] [x_0, x_1, ...] yields [(b_0, x_0), (b_1, x_1), ...], *replicate n x* denotes the list [x, ..., x] of length n, and *nat-of-bool* maps *False* and *True* to *0* and *1*, respectively. We can insert a bit vector in the ith row of a word by

insertll :: *nat* ⇒ α *list* ⇒ α *list list* ⇒ α *list list*
insertll i [] [] = []
insertll i (*a* · *as*) (*bs* · *bss*) = *insertl i a bs* · *insertll i as bss*

where *insertl i a bs* inserts *a* into list *bs* at position *i*. The interaction between *nats-of-boolss* and *insertll* can be characterized by the following theorem:

If ∀ *bs*∈*bss*. *is-alph n bs and* |*bs*| = |*bss*| *and i* ≤ *n then*
nats-of-boolss (*Suc n*) (*insertll i bs bss*) =
insertl i (*nat-of-bools bs*) (*nats-of-boolss n bss*).

Here, *is-alph n xs* means that *xs* is a valid symbol, i.e. the length of *xs* is equal to the number of variables n, |*bs*| and |*bss*| denote the lengths of *bs* and *bss*, respectively, and *bs* ∈ *bss* means that *bs* is a member of the list *bss*. Moreover, *nat-of-bools* is similar to *nats-of-boolss*, with the difference that it works on a single row vector instead of a list of column vectors:

nat-of-bools :: *bool list* ⇒ *nat*
nat-of-bools [] = *0*
nat-of-bools (*b* · *bs*) = *nat-of-bool b* + *2* ∗ *nat-of-bools bs*

Since the input symbols of our automata are bit vectors, it would be rather inefficient to just represent the transition function for a given state as an association list relating bit vectors to successor states. For such a list, the lookup operation would be exponential in the number of variables. When implementing the MONA tool, Klarlund [8] already observed that representing the transition function as a BDD is more efficient. BDDs are represented by the datatype

datatype α *bdd* = *Leaf* α | *Branch* (α *bdd*) (α *bdd*)

The functions *bdd-map* :: (α ⇒ β) ⇒ α *bdd* ⇒ β *bdd* and *bdd-all* :: (α ⇒ *bool*) ⇒ α *bdd* ⇒ *bool* can be defined in a canonical way. The lookup operation, whose runtime is linear in the length of the input vector, has the definition

$bdd\text{-}lookup :: \alpha\ bdd \Rightarrow bool\ list \Rightarrow \alpha$
$bdd\text{-}lookup\ (Leaf\ x)\ bs = x$
$bdd\text{-}lookup\ (Branch\ l\ r)\ (b \cdot bs) = bdd\text{-}lookup\ (\mathsf{if}\ b\ \mathsf{then}\ r\ \mathsf{else}\ l)\ bs$

This operation only returns meaningful results if the height of the BDD is less or equal to the length of the bit vector. We write *bddh n bdd* to mean that the height of *bdd* is less or equal to *n*. Two BDDs can be combined using a binary operator *f* as follows:

$bdd\text{-}binop :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha\ bdd \Rightarrow \beta\ bdd \Rightarrow \gamma\ bdd$
$bdd\text{-}binop\ f\ (Leaf\ x)\ (Leaf\ y) = Leaf\ (f\ x\ y)$
$bdd\text{-}binop\ f\ (Branch\ l\ r)\ (Leaf\ y) =$
$\quad Branch\ (bdd\text{-}binop\ f\ l\ (Leaf\ y))\ (bdd\text{-}binop\ f\ r\ (Leaf\ y))$
$bdd\text{-}binop\ f\ (Leaf\ x)\ (Branch\ l\ r) =$
$\quad Branch\ (bdd\text{-}binop\ f\ (Leaf\ x)\ l)\ (bdd\text{-}binop\ f\ (Leaf\ x)\ r)$
$bdd\text{-}binop\ f\ (Branch\ l_1\ r_1)\ (Branch\ l_2\ r_2) =$
$\quad Branch\ (bdd\text{-}binop\ f\ l_1\ l_2)\ (bdd\text{-}binop\ f\ r_1\ r_2)$

If the two BDDs have different heights, the shorter one is expanded on the fly. The following theorem states that *bdd-binop* yields a BDD corresponding to the pointwise application of *f* to the functions represented by the argument BDDs:

$If\ bddh\ |bs|\ l\ and\ bddh\ |bs|\ r\ then$
$bdd\text{-}lookup\ (bdd\text{-}binop\ f\ l\ r)\ bs = f\ (bdd\text{-}lookup\ l\ bs)\ (bdd\text{-}lookup\ r\ bs).$

2.4 Deterministic Automata

We represent deterministic finite automata (DFAs) by pairs of type *nat bdd list* × *bool list*, where the first and second component denotes the transition function and the set of accepting states, respectively. The states of a DFA are simply natural numbers. Note that we do not mention the start state in the representation of the DFA, since it will always be *0*. Not all pairs of the above type are well-formed DFAs. The two lists must have the same length, and all leaves of the BDDs in the list representing the transition function must be valid states, i.e. be smaller than the length of the two lists. Moreover, the heights of all BDDs must be less or equal to the number of variables *n*, and the set of states must be nonempty. These conditions are captured by the following definition:

$dfa\text{-}is\text{-}node :: dfa \Rightarrow nat \Rightarrow bool$
$dfa\text{-}is\text{-}node\ A \equiv \lambda q.\ q < |fst\ A|$

$wf\text{-}dfa :: dfa \Rightarrow nat \Rightarrow bool$
$wf\text{-}dfa\ A\ n \equiv$
$\quad (\forall\, bdd{\in}fst\ A.\ bddh\ n\ bdd) \wedge$
$\quad (\forall\, bdd{\in}fst\ A.\ bdd\text{-}all\ (dfa\text{-}is\text{-}node\ A)\ bdd) \wedge |snd\ A| = |fst\ A| \wedge 0 < |fst\ A|$

Moreover, the transition function and acceptance condition can be defined by

$dfa\text{-}trans :: dfa \Rightarrow nat \Rightarrow bool\ list \Rightarrow nat$
$dfa\text{-}trans\ A\ q\ bs \equiv bdd\text{-}lookup\ (fst\ A)_{[q]}\ bs$

$dfa\text{-}accepting :: dfa \Rightarrow nat \Rightarrow bool$
$dfa\text{-}accepting\ A\ q \equiv (snd\ A)_{[q]}$

where $xs_{[i]}$ denotes the ith element of list xs. Finally, using the generic functions from §2.2, we can produce variants of these functions tailored to DFAs:

$dfa\text{-}steps\ A \equiv steps\ (dfa\text{-}trans\ A)$
$dfa\text{-}accepts\ A \equiv accepts\ (dfa\text{-}trans\ A)\ (dfa\text{-}accepting\ A)\ 0$
$dfa\text{-}reach\ A \equiv reach\ (dfa\text{-}trans\ A)$

2.5 Nondeterministic Automata

Nondeterministic finite automata (NFAs) are represented by pairs of type *bool list bdd list* × *bool list*. While the second component representing the accepting states is the same as for DFAs, the transition table is now a list of BDDs mapping a state and an input symbol to a finite set of successor states, which we represent as a bit vector. In order for the transition table to be well-formed, the length of the bit vectors representing the sets must be equal to the number of states of the automaton, which coincides with the length of the transition table. This well-formedness condition for the bit vectors is expressed by the predicate

$nfa\text{-}is\text{-}node :: nfa \Rightarrow bool\ list \Rightarrow bool$
$nfa\text{-}is\text{-}node\ A \equiv \lambda qs.\ |qs| = |fst\ A|$

The definition of *wf-nfa* can be obtained from the one of *wf-dfa* by just replacing *dfa-is-node* by *nfa-is-node*. Due to its "asymmetric" type, a transition function of type $nat \Rightarrow bool\ list \Rightarrow bool\ list$ would be incompatible with the abstract functions from §2.2. We therefore lift the function to work on finite sets of natural numbers rather that just single natural numbers. This is accomplished by

$subsetbdd :: bool\ list\ bdd\ list \Rightarrow bool\ list \Rightarrow bool\ list\ bdd \Rightarrow bool\ list\ bdd$
$subsetbdd\ []\ []\ bdd = bdd$
$subsetbdd\ (bdd' \cdot bdds)\ (b \cdot bs)\ bdd =$
$\quad (\mathsf{if}\ b\ \mathsf{then}\ subsetbdd\ bdds\ bs\ (bdd\text{-}binop\ bv\text{-}or\ bdd\ bdd')$
$\quad\ \mathsf{else}\ subsetbdd\ bdds\ bs\ bdd)$

where *bv-or* is the bit-wise or operation on bit vectors, i.e. the union of two finite sets. Using this operation, *subsetbdd* combines all BDDs in the first list, for which the corresponding bit in the second list is *True*. The third argument of *subsetbdd* serves as an accumulator and is initialized with a BDD consisting of only one Leaf containing the empty set, which is the neutral element of *bv-or*:

$nfa\text{-}emptybdd :: nat \Rightarrow bool\ list\ bdd$
$nfa\text{-}emptybdd\ n \equiv Leaf\ (replicate\ n\ False)$

Using *subsetbdd*, the transition function for NFAs can now be defined as follows:

$$\begin{array}{l}\textit{nfa-trans} :: \textit{nfa} \Rightarrow \textit{bool list} \Rightarrow \textit{bool list} \Rightarrow \textit{bool list}\\ \textit{nfa-trans}\ A\ qs\ bs \equiv \textit{bdd-lookup}\ (\textit{subsetbdd}\ (\textit{fst}\ A)\ qs\ (\textit{nfa-emptybdd}\ |qs|))\ bs\end{array}$$

A set of states is accepting iff at least one of the states in the set is accepting:

$$\begin{array}{l}\textit{nfa-accepting}' :: \textit{bool list} \Rightarrow \textit{bool list} \Rightarrow \textit{bool}\\ \textit{nfa-accepting}'\ []\ bs = \textit{False}\\ \textit{nfa-accepting}'\ (a \cdot as)\ [] = \textit{False}\\ \textit{nfa-accepting}'\ (a \cdot as)\ (b \cdot bs) = (a \wedge b \vee \textit{nfa-accepting}'\ as\ bs)\end{array}$$

$$\begin{array}{l}\textit{nfa-accepting} :: \textit{nfa} \Rightarrow \textit{bool list} \Rightarrow \textit{bool}\\ \textit{nfa-accepting}\ A \equiv \textit{nfa-accepting}'\ (\textit{snd}\ A)\end{array}$$

As in the case of DFAs, we can now instantiate the generic functions from §2.2. In order to check whether we can reach an accepting state from the start state, we apply *accepts* to the finite set containing only the state *0*.

$$\begin{array}{l}\textit{nfa-startnode} :: \textit{nfa} \Rightarrow \textit{bool list}\\ \textit{nfa-startnode}\ A \equiv \textit{replicate}\ |\textit{fst}\ A|\ \textit{False}[0 := \textit{True}]\end{array}$$

$$\begin{array}{ll}\textit{nfa-steps}\ A & \equiv \textit{steps}\ (\textit{nfa-trans}\ A)\\ \textit{nfa-accepts}\ A & \equiv \textit{accepts}\ (\textit{nfa-trans}\ A)\ (\textit{nfa-accepting}\ A)\ (\textit{nfa-startnode}\ A)\\ \textit{nfa-reach}\ A & \equiv \textit{reach}\ (\textit{nfa-trans}\ A)\end{array}$$

where $xs[i := y]$ denotes the replacement of the ith element of xs by y.

2.6 Depth First Search

The efficiency of the automata constructions presented in this paper crucially depends on the fact that the generated automata only contain *reachable* states. When implemented in a naive way, the construction of a product DFA from two DFAs having m and n states will lead to a DFA with $m \cdot n$ states, while the construction of a DFA from an NFA with n states will result in a DFA having 2^n states, many of which are unreachable. By using a depth-first search algorithm (DFS) for the generation of the automata, we can make sure that all of their states are reachable. In order to simplify the implementation of the automata constructions, as well as their correctness proofs, the DFS algorithm is factored out into a generic function, whose properties can be proved once and for all. The DFS algorithm is based on a representation of *graphs*, as well as a data structure for storing the nodes that have already been visited. Our version of DFS, which generalizes earlier work by Nishihara and Minamide [13, 10], is designed as an abstract module using the *locale* mechanism of Isabelle, thus allowing the operations on the graph and the node store to be implemented in different ways depending on the application at hand. The module is parameterized by a type α of graph nodes, a type β representing the node store, and the functions

$succs :: \alpha \Rightarrow \alpha\ list$ $\quad$ $ins :: \alpha \Rightarrow \beta \Rightarrow \beta$ $\quad$ $empt :: \beta$
$is\text{-}node :: \alpha \Rightarrow bool$ $\quad$ $memb :: \alpha \Rightarrow \beta \Rightarrow bool$ $\quad$ $invariant :: \beta \Rightarrow bool$

where *succs* returns the list of successors of a node, and the predicate *is-node* describes the (finite) set of nodes. Moreover, *ins x S*, *memb x S* and *empt* correspond to $\{x\} \cup S$, $x \in S$ and $\emptyset$ on sets. The node store must also satisfy an additional *invariant*. Using Isabelle's infrastructure for the definition of functions by well-founded recursion [9], the DFS function can be defined as follows[2]:

$dfs :: \beta \Rightarrow \alpha\ list \Rightarrow \beta$
$dfs\ S\ [] = S$
$dfs\ S\ (x \cdot xs) = (\mathsf{if}\ memb\ x\ S\ \mathsf{then}\ dfs\ S\ xs\ \mathsf{else}\ dfs\ (ins\ x\ S)\ (succs\ x\ @\ xs))$

Note that this function is *partial*, since it may loop when instantiated with *ins*, *memb* and *empt* operators not behaving like their counterparts on sets, or when applied to a list of start values not being valid nodes. However, since *dfs* is *tail recursive*, Isabelle's function definition package can derive the above equations without preconditions, which is crucial for the executability of *dfs*. The central property of *dfs* is that it computes the transitive closure of the successor relation:

If is-node y and is-node x then
$memb\ y\ (dfs\ empt\ [x]) = ((x,\ y) \in (succsr\ succs)^*)$.

where *succsr* turns a successor *function* into a *relation*:

$succsr :: (\gamma \Rightarrow \delta\ list) \Rightarrow \gamma \times \delta \Rightarrow bool$
$succsr\ succs \equiv \{(x,\ y) \mid y \in succs\ x\}$

3 Automata Construction

In this section, we will describe all automata constructions that are used to recursively build automata from formulae in Presburger arithmetic. The simplest one is the *complement*, which we describe in §3.1. It will be used to model *negation*. The *product automaton* construction described in §3.2 corresponds to binary operators such as $\vee$, $\wedge$, and $\longrightarrow$, whereas the more intricate *projection* construction shown in §3.3 is used to deal with existential quantifiers. Finally, §3.4 illustrates the construction of automata corresponding to atomic formulae.

3.1 Complement

The complement construction is straightforward. We only exchange the accepting and the non-accepting states, and leave the transition function unchanged:

[2] We use *dfs S xs* as an abbreviation for *gen-dfs succs ins memb S xs*.

```
prod-succs :: dfa ⇒ dfa ⇒ nat × nat ⇒ (nat × nat) list
prod-succs A B ≡ λ(i, j). add-leaves (bdd-binop Pair (fst A)[i] (fst B)[j]) []

prod-ins :: nat × nat
            ⇒ nat option list list × (nat × nat) list
                ⇒ nat option list list × (nat × nat) list
prod-ins ≡
λ(i, j) (tab, ps). (tab[i := tab[i][j := Some |ps|]], ps @ [(i, j)])

prod-memb :: nat × nat ⇒ nat option list list × (nat × nat) list ⇒ bool
prod-memb ≡ λ(i, j) (tab, ps). tab[i][j] ≠ None

prod-empt :: dfa ⇒ dfa ⇒ nat option list list × (nat × nat) list
prod-empt A B ≡ (replicate |fst A| (replicate |fst B| None), [])

prod-dfs :: dfa ⇒ dfa ⇒ nat × nat ⇒ nat option list list × (nat × nat) list
prod-dfs A B x ≡
gen-dfs (prod-succs A B) prod-ins prod-memb (prod-empt A B) [x]

binop-dfa :: (bool ⇒ bool ⇒ bool) ⇒ dfa ⇒ dfa ⇒ dfa
binop-dfa f A B ≡
let (tab, ps) = prod-dfs A B (0, 0)
in (map (λ(i, j). bdd-binop (λk l. the tab[k][l]) (fst A)[i] (fst B)[j]) ps,
    map (λ(i, j). f (snd A)[i] (snd B)[j]) ps)
```

Fig. 1. Definition of the product automaton

```
negate-dfa :: dfa ⇒ dfa
negate-dfa ≡ λ(t, a). (t, map Not a)
```

A well-formed DFA A will accept a word *bss* iff it is *not* accepted by the DFA produced by *negate-dfa*:

If wf-dfa A n and ∀ *bs*∈*bss. is-alph n bs then*
dfa-accepts (*negate-dfa A*) *bss* = (¬ *dfa-accepts A bss*).

3.2 Product Automaton

Given a binary logical operator $f :: bool \Rightarrow bool \Rightarrow bool$, the product automaton construction is used to build a DFA corresponding to the formula $f\ P\ Q$ from DFAs A and B corresponding to the formulae P and Q, respectively. As suggested by its name, the state space of the product automaton corresponds to the cartesian product of the state spaces of the DFAs A and B. However, as already mentioned in §2.6, not all of the elements of the cartesian product constitute reachable states. We therefore need an algorithm for computing the reachable states of the resulting DFA. Moreover, since the automata framework described in §2.4–2.5 relies on the states to be encoded as natural numbers, we also need

to produce a mapping from *nat* × *nat* to *nat*. All of this can be achieved just by instantiating the abstract DFS framework with suitable functions, as shown in Fig. 1. In this construction, the store containing the visited states is a pair *nat option list list* × (*nat* × *nat*) *list*, where the first component is a matrix denoting a partial map from *nat* × *nat* to *nat*. The second component of the store is a list containing all visited states (i, j). It can be viewed as a map from *nat* to *nat* × *nat*, which is the inverse of the aforementioned map. In order to compute the list of successor states of a state (i, j), *prod-succs* combines the BDDs representing the transition tables of state i of A, and of state j of B using the *Pair* operator, and then collects all leaves of the resulting BDD. The operation *prod-ins* for inserting a state into the store updates the entry at position (i, j) of the matrix *tab* with the number of visited states, and appends (i, j) to the list *ps* of visited states. By definition of DFS, this operation is guaranteed to be applied only if the state (i, j) has not been visited yet, i.e. the corresponding entry in the matrix is *None* and (i, j) is not contained in the list *ps*. We now produce a specific version of DFS called *prod-dfs* by instantiating the generic function from §2.6, and using the list containing just one pair of states as a start value. By induction on *gen-dfs*, we can prove that the matrix and the list computed by *prod-dfs* encodes a bijection between the reachable states (i, j) of the product automaton, and natural numbers k corresponding to the states of the resulting DFA, where k is smaller than the number of reachable states:

If prod-is-node A B x then
$((fst\ (prod\text{-}dfs\ A\ B\ x))_{[i][j]} = Some\ k \wedge dfa\text{-}is\text{-}node\ A\ i \wedge dfa\text{-}is\text{-}node\ B\ j) =$
$(k < |snd\ (prod\text{-}dfs\ A\ B\ x)| \wedge (snd\ (prod\text{-}dfs\ A\ B\ x))_{[k]} = (i, j))$.

The start state x must satisfy a well-formedness condition *prod-is-node*, meaning that its two components must be valid states of A and B. Using this result, as well as the fact that *prod-dfs* computes the transitive closure, we can then show by induction on *bss* that a state m is reachable in the resulting automaton via a sequence of bit vectors *bss* iff the corresponding states s_1 and s_2 are reachable via *bss* in the automata A and B, respectively:

If $\forall$ *bs*∈*bss. is-alph n bs then*
$(\exists\, m.\ dfa\text{-}reach\ (binop\text{-}dfa\ f\ A\ B)\ 0\ bss\ m \wedge$
$(fst\ (prod\text{-}dfs\ A\ B\ (0, 0)))_{[s_1][s_2]} = Some\ m \wedge$
$dfa\text{-}is\text{-}node\ A\ s_1 \wedge dfa\text{-}is\text{-}node\ B\ s_2) =$
$(dfa\text{-}reach\ A\ 0\ bss\ s_1 \wedge dfa\text{-}reach\ B\ 0\ bss\ s_2)$.

Finally, *bdd-binop* produces the resulting product automaton by combining the transition tables of A and B using the mapping from *nat* × *nat* to *nat* computed by *prod-dfs*, and by applying f to the acceptance conditions of A and B. Using the previous theorem, we can prove the correctness statement for this construction:

If wf-dfa A n and wf-dfa B n and $\forall$ *bs*∈*bss. is-alph n bs then*
$dfa\text{-}accepts\ (binop\text{-}dfa\ f\ A\ B)\ bss = f\ (dfa\text{-}accepts\ A\ bss)\ (dfa\text{-}accepts\ B\ bss)$.

3.3 Projection

Using the terminology from §2.3, the automaton for $\exists\, x.\ P$ can be obtained from the one for P by projecting away the row corresponding to the variable x. Since this operation yields an NFA, it is advantageous to first translate the DFA for P into an NFA, which can easily be done by replacing all the leaves in the transition table by singleton sets, and leaving the set of accepting states unchanged. The correctness of this operation called *nfa-of-dfa* is expressed by

If wf-dfa A n and $\forall$ *bs*$\in$*bss. is-alph n bs then*
nfa-accepts (*nfa-of-dfa A*) *bss* = *dfa-accepts A bss*.

Given a BDD representing the transition table of a particular state of an NFA, we can project away the ith variable by combining the two children BDDs of the branches at depth i using the *bv-or* operation:

quantify-bdd :: *nat* $\Rightarrow$ *bool list bdd* $\Rightarrow$ *bool list bdd*
quantify-bdd i (*Leaf q*) = *Leaf q*
quantify-bdd 0 (*Branch l r*) = *bdd-binop bv-or l r*
quantify-bdd (*Suc i*) (*Branch l r*) =
  Branch (*quantify-bdd i l*) (*quantify-bdd i r*)

To produce the NFA corresponding to the quantified formula, we just map this operation over the transition table:

quantify-nfa :: *nat* $\Rightarrow$ *nfa* $\Rightarrow$ *nfa*
quantify-nfa i $\equiv$ λ(*bdds*, *as*). (*map* (*quantify-bdd i*) *bdds*, *as*)

Due to its type, we could apply this function repeatedly to quantify over several variables in one go. The correctness of this construction is summarized by

If wf-nfa A (*Suc n*) *and i* $\leq$ *n and* $\forall$ *bs*$\in$*bss. is-alph n bs then*
nfa-accepts (*quantify-nfa i A*) *bss* =
($\exists$ *bs. nfa-accepts A* (*insertll i bs bss*) $\wedge$ |*bs*| = |*bss*|).

This means that the new NFA accepts a list *bss* of column vectors iff the original NFA accepts the list obtained from *bss* by inserting a suitable row vector *bs* representing the existential witness. Matters are complicated by the additional requirement that the word accepted by the new NFA must have the same length as the witness. This requirement can be satisfied by appending zero vectors to the end of *bss*, which does not change its interpretation. Since the other constructions (in particular the complement) only work on DFAs, we turn the obtained NFA into a DFA by applying the usual *subset construction*. The central idea is that each set of states produced by *nfa-steps* can be viewed as a state of a new DFA. As mentioned in §2.6, not all of these sets are reachable from the initial state of the NFA. Similar to the *product* construction, the algorithm for computing the reachable sets shown in Fig. 2 is an instance of the general DFS framework. The node store is now a pair of type *nat option bdd* $\times$ *bool list list*, where the

first component is a BDD representing a partial map from finite sets (encoded as bit vectors) to natural numbers, and the second component is the list of visited states representing the inverse map. To insert new entries into a BDD, we use

```
bddinsert :: α bdd ⇒ bool list ⇒ α ⇒ α bdd
bddinsert (Leaf a) [] x = Leaf x
bddinsert (Leaf a) (w · ws) x =
  (if w then Branch (Leaf a) (bddinsert (Leaf a) ws x)
   else Branch (bddinsert (Leaf a) ws x) (Leaf a))
bddinsert (Branch l r) (w · ws) x =
  (if w then Branch l (bddinsert r ws x) else Branch (bddinsert l ws x) r)
```

The computation of successor states in *subset-succs* and the transition relation in *det-nfa* closely resembles the definition of *nfa-trans* from §2.5. Using the fact that *subset-dfs* computes a bijection between finite sets and natural numbers, we can prove the correctness theorem for *det-nfa*:

```
If wf-nfa A n and ∀ bs∈bss. is-alph n bs then
dfa-accepts (det-nfa A) bss = nfa-accepts A bss.
```

Recall that the automaton produced by *quantify-nfa* will only accept words with a sufficient number of trailing zero column vectors. To get a DFA that also accepts words without trailing zeros, we mark all states as accepting, from which an accepting state can be reached by reading only zeros. This construction, which is sometimes referred to as the *right quotient*, can be characterized as follows:

```
If wf-dfa A n and ∀ bs∈bss. is-alph n bs then
dfa-accepts (rquot A n) bss = (∃ m. dfa-accepts A (bss @ zeros m n)).
```

where *zeros m n* produces a word consisting of m zero vectors of size n.

3.4 Diophantine (In)Equations

We now come to the construction of DFAs for atomic formulae, namely Diophantine (in)equations. For this purpose, we use a method due to Boudet and Comon [3]. The key observation is that *xs* is a solution of a Diophantine equation iff it is a solution modulo *2*, and the quotient of *xs* and *2* is a solution of another equation with the same coefficients, but with a different right-hand side:

```
(eval-dioph ks xs = l) =
(eval-dioph ks (map (λx. x mod 2) xs) mod 2 = l mod 2 ∧
 eval-dioph ks (map (λx. x div 2) xs) =
 (l − eval-dioph ks (map (λx. x mod 2) xs)) div 2)
```

In other words, the states of the DFA accepting the solutions of the equation correspond to the right-hand sides reachable from the initial right-hand side l, which will again be computed using the DFS algorithm. To ensure termination of DFS, it is crucial that the reachable right-hand sides m are bounded:

```
subset-succs :: nfa ⇒ bool list ⇒ bool list list
subset-succs A qs ≡ add-leaves (subsetbdd (fst A) qs (nfa-emptybdd |qs|)) []

subset-ins :: bool list
              ⇒ nat option bdd × bool list list
                 ⇒ nat option bdd × bool list list
subset-ins qs ≡ λ(bdd, qss). (bddinsert bdd qs (Some |qss|), qss @ [qs])

subset-memb :: bool list ⇒ nat option bdd × bool list list ⇒ bool
subset-memb qs ≡ λ(bdd, qss). bdd-lookup bdd qs ≠ None

subset-empt :: nat option bdd × bool list list
subset-empt ≡ (Leaf None, [])

subset-dfs :: nfa ⇒ bool list ⇒ nat option bdd × bool list list
subset-dfs A x ≡
gen-dfs (subset-succs A) subset-ins subset-memb subset-empt [x]

det-nfa :: nfa ⇒ dfa
det-nfa A ≡
let (bdd, qss) = subset-dfs A (nfa-startnode A)
in (map (λqs. bdd-map (λqs. the (bdd-lookup bdd qs))
                 (subsetbdd (fst A) qs (nfa-emptybdd |qs|)))
    qss,
   map (nfa-accepting A) qss)
```

Fig. 2. Definition of the subset construction

$$\textit{If } |m| \leq \textit{max } |l| \ (\textstyle\sum k{\leftarrow}\textit{ks}.\ |k|) \textit{ then}$$
$$|(m - \textit{eval-dioph ks } (\textit{map } (\lambda x.\ x \textit{ mod } 2)\ \textit{xs}))\ \textit{div } 2| \leq \textit{max } |l| \ (\textstyle\sum k{\leftarrow}\textit{ks}.\ |k|).$$

By instantiating the abstract *gen-dfs* function, we obtain a function

$$\textit{dioph-dfs} :: \textit{nat} \Rightarrow \textit{int list} \Rightarrow \textit{int} \Rightarrow \textit{nat option list} \times \textit{int list}$$

that, given the number of variables, the coefficients, and the right-hand side, computes a bijection between reachable right-hand sides and natural numbers:

$$\begin{aligned}&((\textit{fst } (\textit{dioph-dfs } n\ \textit{ks } l))_{[\textit{int-to-nat-bij } m]} = \textit{Some } k \wedge\\ &\ |m| \leq \textit{max } |l| \ (\textstyle\sum k{\leftarrow}\textit{ks}.\ |k|)) =\\ &(k < |\textit{snd } (\textit{dioph-dfs } n\ \textit{ks } l)| \wedge (\textit{snd } (\textit{dioph-dfs } n\ \textit{ks } l))_{[k]} = m)\end{aligned}$$

The first component of the pair returned by *dioph-dfs* can be viewed as a partial map from integers to natural numbers, where *int-to-nat-bij* maps negative and non-negative integers to odd and even list indices, respectively. As shown in Fig. 3, the transition table of the DFA is constructed by *eq-dfa* as follows: if the current state corresponds to the right-hand side j, and the DFA reads a bit vector *xs* satisfying the equation modulo *2*, then the DFA goes to the state corresponding to the new right-hand side $(j - \textit{eval-dioph ks xs})\ \textit{div } 2$, otherwise

```
eq-dfa :: nat ⇒ int list ⇒ int ⇒ dfa
eq-dfa n ks l ≡
let (is, js) = dioph-dfs n ks l
in (map (λj. make-bdd
              (λxs. if eval-dioph ks xs mod 2 = j mod 2
                    then the is_[int-to-nat-bij ((j − eval-dioph ks xs) div 2)] else |js|)
              n [])
    js @
    [Leaf |js|],
    map (λj. j = 0) js @ [False])
```

Fig. 3. Definition of the automata for Diophantine equations

it goes to an error state, which is the last state in the table. To produce a BDD containing the successor states for all bit vectors of length n, we use the function

```
make-bdd :: (nat list ⇒ α) ⇒ nat ⇒ nat list ⇒ α bdd
make-bdd f 0 xs = Leaf (f xs)
make-bdd f (Suc n) xs =
  Branch (make-bdd f n (xs @ [0])) (make-bdd f n (xs @ [1]))
```

The key property of *eq-dfa* states that for every right-hand side m reachable from l, the state reachable from m via a word *bss* is accepting iff the list of natural numbers denoted by *bss* satisfies the equation with right-hand side m:

If $(l, m) \in (succsr\ (dioph\text{-}succs\ n\ ks))^*$ *and* $\forall\, bs{\in}bss.\ is\text{-}alph\ n\ bs$ *then*
dfa-accepting $(eq\text{-}dfa\ n\ ks\ l)$
$(dfa\text{-}steps\ (eq\text{-}dfa\ n\ ks\ l)\ (the\ (fst\ (dioph\text{-}dfs\ n\ ks\ l))_{[int\text{-}to\text{-}nat\text{-}bij\ m]})\ bss) =$
$(eval\text{-}dioph\ ks\ (nats\text{-}of\text{-}boolss\ n\ bss) = m)$.

Here, *dioph-succs n ks* returns a list of up to 2^n successor states reachable from a given state by reading a single column vector of size n. The proof of the above property is by induction on *bss*, where the equation given at the beginning of this section is used in the induction step. The correctness property of *eq-dfa* can then be obtained from this result as a simple corollary:

If $\forall\, bs{\in}bss.\ is\text{-}alph\ n\ bs$ *then*
$dfa\text{-}accepts\ (eq\text{-}dfa\ n\ ks\ l)\ bss = (eval\text{-}dioph\ ks\ (nats\text{-}of\text{-}boolss\ n\ bss) = l)$.

Diophantine inequations can be treated in a similar way.

4 The Decision Procedure

We now have all the machinery in place to write a decision procedure for Presburger arithmetic. A formula can be transformed into a DFA by the following function:

$dfa\text{-}of\text{-}pf :: nat \Rightarrow pf \Rightarrow dfa$
$dfa\text{-}of\text{-}pf\ n\ (Eq\ ks\ l) = eq\text{-}dfa\ n\ ks\ l$
$dfa\text{-}of\text{-}pf\ n\ (Le\ ks\ l) = ineq\text{-}dfa\ n\ ks\ l$
$dfa\text{-}of\text{-}pf\ n\ (Neg\ p) = negate\text{-}dfa\ (dfa\text{-}of\text{-}pf\ n\ p)$
$dfa\text{-}of\text{-}pf\ n\ (And\ p\ q) = binop\text{-}dfa\ (\wedge)\ (dfa\text{-}of\text{-}pf\ n\ p)\ (dfa\text{-}of\text{-}pf\ n\ q)$
$dfa\text{-}of\text{-}pf\ n\ (Or\ p\ q) = binop\text{-}dfa\ (\vee)\ (dfa\text{-}of\text{-}pf\ n\ p)\ (dfa\text{-}of\text{-}pf\ n\ q)$
$dfa\text{-}of\text{-}pf\ n\ (Imp\ p\ q) = binop\text{-}dfa\ (\longrightarrow)\ (dfa\text{-}of\text{-}pf\ n\ p)\ (dfa\text{-}of\text{-}pf\ n\ q)$
$dfa\text{-}of\text{-}pf\ n\ (Exist\ p) =$
$\quad rquot\ (det\text{-}nfa\ (quantify\text{-}nfa\ 0\ (nfa\text{-}of\text{-}dfa\ (dfa\text{-}of\text{-}pf\ (Suc\ n)\ p)))\ n$
$dfa\text{-}of\text{-}pf\ n\ (Forall\ p) = dfa\text{-}of\text{-}pf\ n\ (Neg\ (Exist\ (Neg\ p)))$

By structural induction on formulae, we can show the correctness theorem

$If\ \forall\, bs{\in}bss.\ is\text{-}alph\ n\ bs\ \ then$
$dfa\text{-}accepts\ (dfa\text{-}of\text{-}pf\ n\ p)\ bss = eval\text{-}pf\ p\ (nats\text{-}of\text{-}boolss\ n\ bss).$

Note that a closed formula is valid iff the start state of the resulting DFA is accepting, which can easily be seen by letting $n = 0$ and $bss = []$. Most cases of the induction can be proved by a straightforward application of the correctness results from §3. Unsurprisingly, the only complicated case is the one for the existential quantifier, which we will now examine in more detail. In this case, the left-hand side of the correctness theorem is

$dfa\text{-}accepts$
$(rquot\ (det\text{-}nfa\ (quantify\text{-}nfa\ 0\ (nfa\text{-}of\text{-}dfa\ (dfa\text{-}of\text{-}pf\ (Suc\ n)\ p)))\ n)\ bss$

which, according to the correctness statement for *rquot*, is equivalent to

$\exists\, m.\ dfa\text{-}accepts\ (det\text{-}nfa\ (quantify\text{-}nfa\ 0\ (nfa\text{-}of\text{-}dfa\ (dfa\text{-}of\text{-}pf\ (Suc\ n)\ p))))$
$(bss\ @\ zeros\ m\ n)$

By correctness of *det-nfa*, *quantify-nfa* and *nfa-of-dfa*, this is the same as

$\exists\, m\ bs.\ dfa\text{-}accepts\ (dfa\text{-}of\text{-}pf\ (Suc\ n)\ p)\ (insertll\ 0\ bs\ (bss\ @\ zeros\ m\ n)) \wedge$
$\qquad |bs| = |bss| + |zeros\ m\ n|$

Using the induction hypothesis, this can be rewritten to

$\exists\, m\ bs.\ eval\text{-}pf\ p\ (nats\text{-}of\text{-}boolss\ (Suc\ n)\ (insertll\ 0\ bs\ (bss\ @\ zeros\ m\ n))) \wedge$
$\qquad |bs| = |bss| + |zeros\ m\ n|$

According to the properties of *nats-of-boolss* from §2.3, this can be recast as

$\exists\, m\ bs.\ eval\text{-}pf\ p\ (nat\text{-}of\text{-}bools\ bs \cdot nats\text{-}of\text{-}boolss\ n\ bss) \wedge |bs| = |bss| + m$

which is obviously equivalent to the right-hand side of the correctness theorem

$\exists\, x.\ eval\text{-}pf\ p\ (x \cdot nats\text{-}of\text{-}boolss\ n\ bss)$

since we can easily produce suitable instantiations for m and bs from x.

5 Conclusion

First experiments with the algorithm presented in §4 show that it can compete quite well with the standard decision procedure for Presburger arithmetic available in Isabelle. Even without minimization, the DFA for the stamp problem from §2.1 has only 6 states, and can be constructed in less than a second. The following table shows the size of the DFAs (i.e. the number of states) for all subformulae of the stamp problem. Thanks to the DFS algorithm, they are much smaller than the DFAs that one would have obtained using a naive construction:

<table>
<tr><td rowspan="2">Forall
6</td><td rowspan="2">Imp
15</td><td>Exist
13</td><td>Exist
9</td><td colspan="2">Eq [5, 3, −1] 0
9</td></tr>
<tr><td colspan="4">Le [−1] −8
5</td></tr>
</table>

The next step is to formalize a minimization algorithm, e.g. along the lines of Constable et al. [6]. We also intend to explore other ways of constructing DFAs for Diophantine equations, such as the approach by Wolper and Boigelot [15], which is more complicated than the one shown in §3.4, but can directly deal with variables over the integers rather than just natural numbers. To improve the performance of the decision procedure on large formulae, we would also like to investigate possible optimizations of the simple representation of BDDs presented in §2.3. Verma [14] describes a formalization of reduced ordered BDDs with *sharing* in Coq. To model sharing, Verma's formalization is based on a *memory* for storing BDDs. Due to their dependence on the *memory*, algorithms using this kind of BDDs are no longer *purely functional*, which makes reasoning about them substantially more challenging. Finally, we also plan to extend our decision procedure to cover WS1S, and use it to tackle some of the circuit verification problems described by Basin and Friedrich [1].

Acknowledgements We would like to thank Tobias Nipkow for suggesting this project and for numerous comments, Clemens Ballarin and Markus Wenzel for answering questions concerning locales, and Alex Krauss for help with well-founded recursion and induction schemes.

References

1. D. Basin and S. Friedrich. Combining WS1S and HOL. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, February 2000.
2. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: TYPES'2000*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

3. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Proceedings*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.
4. S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer-Verlag, 1997.
5. A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41:33–59, 2008.
6. R. L. Constable, P. B. Jackson, P. Naumov, and J. Uribe. Constructively formalizing automata theory. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.
7. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz`.
8. N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11th International Workshop, CSL '97, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 311–326. Springer-Verlag, 1998.
9. A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer-Verlag, 2006.
10. Y. Minamide. Verified decision procedures on context-free grammars. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 173–188. Springer-Verlag, 2007.
11. T. Nipkow. Verified lexical analysis. In J. Grundy and M. C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1998.
12. T. Nipkow. Linear quantifier elimination. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 18–33. Springer-Verlag, 2008.
13. T. Nishihara and Y. Minamide. Depth first search. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Depth-First-Search.shtml`, June 2004. Formal proof development.
14. K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In J. He and M. Sato, editors, *6th Asian Computing, Proceedings*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer-Verlag, 2000.
15. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2000.