CHECKING VALIDITIES AND PROOFS WITH CVC AND FLEA

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Aaron Stump

September 2002

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David L. Dill
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Zohar Manna

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell

Approved for the University Committee on Graduate Studies:

Abstract

In automated reasoning, *decision procedures* are algorithms capable of reporting whether or not a formula is logically valid. Decision procedures are fundamental tools for formal verification. This thesis describes solutions to theoretical and engineering problems arising in the development of the CVC ("Cooperating Validity Checker") decision procedure. CVC implements a framework that allows independent decision procedures for the quantifier-free consequences of different logical theories to be combined. Assuming certain properties of those decision procedures, the combination results in a sound and complete decision procedure for the quantifier-free consequences of the combined theory. There are currently decision procedures implemented for theories of arrays, linear real arithmetic, and inductive data types.

The thesis begins with an overview of the internals of CVC. Then CVC's array decision procedure is described. This is the first published procedure for its theory, which subsumes similar theories previously decided in the literature. Next, an approach is described for handling partial functions. The approach is based on a theorem reducing validity in a logic of partial functions to validity in classical logic. Then proof production in CVC is considered. CVC has the ability to produce a proof for every formula it reports to be valid. These proofs are produced in a proof meta-language based on the Edinburgh Logical Framework (LF). The proofs can be checked by a proof checker called flea. Several new language features are supported in flea which make it much easier to produce proofs from CVC. Also, flea implements new optimizations to the basic LF type-checking algorithm to be able to check the large proofs produced by CVC for large input formulas. Empirical results showing the benefits of the optimizations are presented. Finally, several uses of proofs beyond certifying correctness of CVC's results are described, including a technique which extracts information from proofs to feed back to CVC's propositional reasoning engine, the Chaff SAT solver, for improved performance.

Acknowledgements

This work was supported in part by a National Science Foundation Graduate Fellowship. Support was also provided by NSF contract CCR-9806889-002 and ARPA/AirForce contract F33615-00-C-1693. I thank first of all my advisor, Prof. Dill, for his support, criticism, and instruction, which have been very profitable for my development as a researcher. I thank the members of the reading committee, Prof. Dill, Prof. Manna, and Prof. Mitchell, for their time and feedback. I also thank the following other professors from whom I have learned much at Stanford: Prof. Feferman (Mathematics), Prof. Lam, Prof. Mints (Philosophy), Prof. Pratt, and Prof. van Benthem (CSLI). I also thank Prof. Pfenning at Carnegie Mellon for his generous and patient help and advice with technical and more general matters. I am grateful to Prof. Schwichtenberg of the Ludwig Maximilian University, Munich, for his introductory proof theory course during a visit to Stanford, and for other advice and support. I am grateful to my fellow students in Prof. Dill's research group, and especially to the students in Prof. Manna's research group, with whom I had many interesting technical and other discussions. I particularly acknowledge the very many profitable and fun conversations I had with my officemate Clark Barrett in my time at Stanford. I also thank Iliano Cervesato and Nikolaj Björner for introducing me to many fascinating technical matters. I thank my friends from Stanford and Cornell for their support during my time at Stanford. I thank also the staff at Peninsula Bible Church for religious support my first two years at Stanford. I especially thank the ministers and congregation of Trinity Lutheran Church for much support in the last two years. I owe a deep debt of thanks to my mother, father, sister, and brother. I dedicate my work at Stanford "$\mu o \nu \omega\ \ \sigma o \phi \omega\ \ \theta \epsilon \omega$": to God only wise.

Contents

List of Figures

Chapter 1

Introduction

In automated reasoning, *decision procedures* are algorithms capable of reporting whether or not a formula is logically valid. Decision procedures have attracted much attention for use in formal verification of software, hardware, and protocols. They play a crucial role in verification based on interactive theorem provers such as PVS [ORS92] and NuPRL [CtPg86], where decidable subgoals that arise in proofs can be automatically discharged by decision procedures, thus reducing the burden on the user. They have also been used in more direct approaches to verification, where verification problems are reduced to validity checking problems (e.g., [SJD98]). Propositional SAT solvers like Chaff [MMZ$^+$01] and GRASP [MSS99] have enjoyed considerable recent success in this approach (e.g., [BLM01, CBRZ01, KS00, GK99]).

For verification, it is highly desirable to have decision procedures that can reason about the various data types that occur in system descriptions. Such data types include integers and (finite) reals, arrays, bitvectors, records, tuples, lists, trees, streams, and others. Even given decision procedures for decidable fragments of first-order theories of these sorts of data types, it is not *a priori* clear how to combine those decision procedures to reason about mixed data structures like records of arrays of trees of reals. Such data structures abound in system descriptions. Two approaches have been developed in the literature for combining decision procedures for different first-order classical theories: the Nelson-Oppen approach [NO79] and the Shostak approach [Sho84]. Both approaches aim to combine decision procedures for quantifier-free fragments of different classical first-order theories, where the theories do not share any function or predicate symbols. Both approaches have received renewed attention recently [TH96, BT97, CLS96, RS01, Bjo99, BDL96, BDS00, BDS02a]. The Nelson-Oppen approach is more general, in the sense that it places fewer restrictions on the kinds of decision procedures that can be combined than does the Shostak approach.

Implementing cooperating decision procedures based on either the Nelson-Oppen or the Shostak approach is a significant engineering effort. Each decision procedure that is combined can be a fairly complex piece of code. The combination of the decision procedures is thus quite complicated indeed. For use in verification, one may easily be concerned about the correctness of the results obtained from such complex pieces of software. An approach advocated recently by Necula and Lee in the context of proof-carrying code [Nec97] is to have theorem provers produce proofs for valid formulas, where the proofs can be easily and independently checked [NL00]. Whenever the cooperating decision procedure reports that a formula is valid, the proof it produces may be checked to confirm that result.

Existing tools that combine decision procedures include:

- ICS at SRI (http://www.icansolve.org)
- Simplify at Compaq SRC (http://research.compaq.com/SRC/esc/Simplify.html)
- STeP at Stanford (http://www-step.stanford.edu/)
- SVC at Stanford (http://verify.stanford.edu/SVC/)
- Vampyre at Berkeley (http://www-cad.eecs.berkeley.edu/ rupak/Vampyre/)

Of these systems, only Vampyre and Simplify are based on the more general Nelson-Oppen approach; the others are all based on the Shostak approach. Only Vampyre produces independently checkable proofs of valid formulas. SVC is probably the only system targetted at very large formulas (on the order of megabytes long in ASCII text).

This thesis centers around CVC ("a Cooperating Validity Checker"), a new cooperating decision procedure that has the best of all the features:

- it is based on a version of the more general Nelson-Oppen combination technique
- it produces independently checkable proofs of formulas it claims to be valid

- it is targetted at large (megabytes long) formulas

CVC also incorporates the highly efficient Chaff SAT solver [MMZ$^+$01] for its propositional reasoning. CVC currently includes subsidiary decision procedures for a theory of arrays, linear real arithmetic, and abstract data types like records, tuples, lists, and trees. A decision procedure for arbitrary uninterpreted functions and predicates is also included. CVC's input language is typed. There are built-in primitive types like the type for real numbers. The user may also declare uninterpreted types and abstract data types. Typed constants, functions, and predicates may also be declared. Their types are either primitive types or built from types using type constructors like cartesian product, array type constructor, and the (first-order) function space constructor. The user can then query whether formulas over the declared and built-in symbols (like $+$ and $<$) of the theories are valid or not. Detailed documentation about CVC's input language is available on CVC's web site, http://verify.stanford.edu/CVC. CVC has been freely available at that site since November, 2001.

The thesis begins with a brief high-level overview of the internals of CVC. Then CVC's array decision procedure is described (Chapter 3). This decision procedure is the first published procedure for its theory, which subsumes all other theories previously decided in the literature. Chapter 4 describes an approach to handling partial functions in decision procedures. Examples of partial functions in CVC are division, which is considered undefined if the divisor is zero; and data type selectors like *car*, which is undefined if given *NULL*. A theorem is proved showing the correctness of CVC's approach to handling partial functions, and applications of the theorem are given. Chapter 5 explains how proofs are produced from CVC. A proof-producing version of the union-find data structure, which is used to maintain a database of known equalities, is presented. A proof-producing decision procedure for linear real arithmetic is also described. This chapter also introduces the **flea** proof checker, which has been developed to check proofs produced from CVC. The proofs are produced in an extension of the Edinburgh Logical Framework (LF) [HHP93]. The extended version of LF makes it convenient to produce proofs in the presence of arity-polymorphic operators like n-ary addition and n-ary tuple formation, which can take in any number of arguments. Chapter 5.5 describes other uses for proofs produced by CVC, the most important of which is a technique for determining the subset of assumptions relevant to a given inconsistency detected by CVC. This information can be fed back to the

SAT solver, resulting in improved performance. Chapter 6 presents optimizations to proof checking in LF. These optimizations are required to check the large proofs produced by CVC for large valid formulas. Correctness of the main optimization is established using basic techniques from proof theory. Chapter 7 gives further selected details about the implementation of CVC. In order to improve usefulness of the thesis, each chapter is mostly self-contained.

The end result of this work is a general, efficient, proof-producing decision procedure suitable for use in verification. The proofs produced can be checked efficiently by a relatively small proof checker. Parts of this thesis have been accepted for publication [SBD02, SBDL01, SD02a, SD02b]. Note that the implementation of the Nelson-Oppen approach used and the techniques for incorporating Chaff into CVC are the topics of another thesis [Bar02] (discussed also in [BDS02b] and [BDS00]).

Chapter 2

Overview of CVC

This chapter gives a brief introduction to the implementation of CVC. CVC has two input languages that may be used to describe a formula whose logical validity is to be checked. One language is the presentation language. This language has customary mathematical and logical notations for the various operators. The presentation language is intended to be convenient for humans to use. The second language is CVC's internal language, to which expressions in the presentation language are reduced after they are parsed into the system. Some constructs in the presentation language do not exist in the internal language; they are removed during translation from the presentation to the internal language. The syntax for the internal language uses a fully parenthesized prefix notation. This is easier for tools targetting CVC to generate than the syntax for the presentation language. In this section, the syntax used for CVC expressions will be that of the presentation language.

2.1 CVC's type system

CVC's input language is typed. So for all but trivial validity checking problems, the types of the variables occurring in a formula must be declared before that formula can be processed. CVC's type system is essentially that of multi-sorted first-order logic, with a few extensions.

Inductive datatypes

CVC allows inductive datatypes to be declared with a declaration like the following:

```
alist : TYPE = DATATYPE cons (car : a, cdr : alist),
                        null
                END;
```

This states that *alist* is a new datatype with constructors *cons* and *null*. Selectors *car* and *cdr* are specified for *cons*. Testers *cons?* and *null?* are automatically added for each constructor.

Inductive datatypes are a significant logical addition. They are not a significant addition to the type system, however. CVC's type system restricts polymorphism to built-in operators. Hence, it is not possible for the user to declare a polymorphic type of lists of any type of data. Furthermore, inductive datatypes are compared by name equivalence. Anonymous datatypes are not permitted. So from the point of view of CVC's type checker, an inductive datatype behaves just like a primitive type like *REAL*. The situation would be more complex if inductive datatypes were implemented using a least fixed point type constructor.

CVC's presentation language has special notation for record and tuple types, even though these may be seen as special cases of inductive datatypes.

Polymorphic symbols

Certain function and predicate symbols in CVC's language are polymorphic. The function symbols *read* and *write* from the theory of arrays, for example, operate on arrays holding arbitrary data. The equality symbol is polymorphic; the two sides of the equality are merely required to be of the same type. CVC also has a polymorphic if-then-else operator. The then- and else-parts are required to have the same types.

As mentioned above, polymorphism is currently restricted in CVC to built-in operations. So type-checking for a general polymorphic type system need not be implemented. CVC's implementation allows special purpose code to be plugged into the type checker by different theories to support any built-in polymorphic operations they may have.

2.1.1 Boolean operations

From a logical perspective, CVC's input language is first-order. It is crucial for the proper functioning of the cooperating decision procedures inside CVC that all expressions which ultimately reach them during the processing of a large formula should be first-order. Technically, this would rule out functions and predicates which take in booleans as arguments.

Such operations are essential, however, for modeling systems like hardware designs. The problem is finessed in CVC by adding a new datatype for boolean terms. Type declarations are pre-processed so that operations declared by the user to take booleans as inputs are declared internally to take boolean terms. Input formulas are also pre-processed to replace every boolean argument B to a predicate or function by *IF B THEN TRUE_TERM ELSE FALSE_TERM*, where *TRUE_TERM* and *FALSE_TERM* are the two constructors for the datatype of boolean terms. Other parts of CVC's infrastructure are aware of if-then-else, and do not allow it to propagate to the core cooperating decision procedures. This approach allows users to declare functions and predicate symbols taking booleans as arguments, without requiring a modification of the core infrastructure for cooperating decision procedures.

2.2 Cooperating decision procedures

The framework for cooperating decision procedures implemented in CVC allows independent decision procedures for different logical theories to be combined to decide the union of the theories. Certain restrictions must be satisfied for this to be possible, the most basic of which is that the theories decided may not share any function or predicate symbols, except for the equality symbol. Hence, if we view equality as a logical symbol, every non-logical symbol belongs to exactly one theory.

Each subsidiary decision procedure takes in a set of *literals* in the language of its theory. Literals are atomic formulas like $x < 3$ or $a[i] = b[j]$, or their negations, not containing any occurrences of if-then-else. The literals are accepted by the subsidiary decision procedure one at a time. The decision procedure is responsible for notifying the framework if it detects that the set of literals it has been given is unsatisfiable. Giving a literal to a decision procedure to add to this set is called *asserting* the literal to the decision procedure.

The framework for cooperating decision procedures for theories $T_1, \ldots, T_n$ allows literals in the language of $T_1 \cup \ldots \cup T_n$ to be asserted to it. Like one of its subsidiary decision procedures, the framework is responsible for testing whether or not the set of literals asserted to it so far is unsatisfiable. The set of literals currently added is called the *current logical context* of the framework. When a literal that happens to be in the language of exactly one theory is asserted to the cooperating decision procedures, the assertion of that literal is propagated to the decision procedure for that theory. When a literal containing symbols from more than one theory is asserted, however, the assertion of that literal is propagated to the decision procedure for the theory containing the top-most left-most non-logical symbol. For example, $\neg(\mathit{car(a[i])} = 3 * z)$ would go to the decision procedure for the theory of inductive data types, since *car* is the top-most left-most non-logical symbol in that literal. Each subsidiary decision procedure ignores, so to speak, applications of function symbols that are not in their theory. Such terms are just treated by the subsidiary decision procedure as variables. This allows the decision procedure to handle the literal as if it contained only symbols from the theory decided by the decision procedure. In addition to propagating the assertion of the literal to the decision procedure owning the top-most left-most non-logical symbol, the framework also notifies other decision procedures about any terms from their theories which occur as arguments to a symbol from another theory. Such a term is considered shared between the decision procedure owning its function symbol and the decision procedure owning the symbol that is being applied to the term. One of the obligations of every subsidiary decision procedure in CVC is to propagate any equalities between shared terms that are entailed by the set of literals asserted to that decision procedure so far. See [Bar02] for full details on the framework for cooperating decision procedures implemented in CVC.

2.3 Propositional reasoning

To decide arbitrary quantifier-free formulas which may contain conjunctions, disjunctions, negations, and if-then-else expressions, CVC needs to use propositional reasoning to reduce the problem to a set of subproblems in the form required by the cooperating decision procedures. To do this propositional reasoning, CVC has two SAT solvers built in. The first is a naive Davis-Putnam procedure, which simply searches the input formula top-down for atomic subformulas to split on. The formula is simplified after asserting that the atomic subformula is false (and possibly then asserting it to be true). The assertion of the splitter or its negation propagates to the cooperating decision procedures. The second SAT solver is Chaff, one of the most efficient SAT solvers currently available [MMZ$^+$01]. The integration of Chaff with the cooperating decision procedure is similar in spirit to that of the naive SAT solver, but it is more complicated both from an engineering perspective and because a tighter integration is possible than with the naive SAT solver. The tighter integration is based on exporting small conflict clauses to Chaff from the cooperating decision procedures. See Section 5.5.3 below for more about this.

The SAT solvers both rely on the fact that the state of the cooperating decision procedure is backtrackable. Backtracking in CVC is mediated by an interface class called Context_Manager. This interface class has the following methods:

```
Context *   CreateContext();
void        DeleteContext(Context *context);

Context *   GetCurrentContext();
void        SetCurrentContext(Context *context);

unsigned    GetCurrentScopeLevel();
unsigned    IncrementCurrentScopeLevel(unsigned offset);
void        DeleteScopesAfterScopeLevel(unsigned sl);
```

The first two methods create and delete Contexts. A Context holds all the backtrackable state in the system. Multiple Contexts can exist at once, each holding state for the system in a parallel universe, so to speak. The method SetCurrentContext() allows the system to switch between these parallel universes. The implementation is such that this switch happens in constant time. The last group of methods allows scopes to be pushed and popped in the current Context. Changes made to state in a given scope will be undone when that scope is popped. A call to DeleteScopesAfterScopeLevel() causes the state of the system to revert to what it was when the scope with the given level was first pushed. The ability to modify the state of the system and incrementally undo those modifications is crucial for the integration of the cooperating decision procedures with the SAT solver. The ability to switch between parallel universes is not, but situations have arisen in implementing CVC where this very powerful feature has been quite convenient.

Chapter 3

A decision procedure for a theory of arrays

This chapter describes CVC's decision procedure for a theory of arrays. For applications in formal verification and program analysis, it is very important to be able to reason about arrays. Arrays are ubiquitous in software. Structures like memories and register files in hardware or message queues in protocols can be modeled as arrays. Furthermore, a decision procedure for a theory of arrays is also of value for theorem-provers. The PVS theorem-prover [ORS92] has an undocumented decision procedure for a theory of arrays [Rue00], and HOL has some automatic support for a theory of arrays via a library for finite partial functions [CS95].

Two kinds of array theories have been studied previously. Extensional theories require that if two arrays store the same value at index i, for each index i, then the arrays must be the same. Non-extensional theories do not make this requirement. The paper [SBDL01], which first described the results of this chapter, was the first to present a procedure for checking satisfiability of arbitrary quantifier-free formulas in an extensional theory of arrays and prove its correctness.

3.1 Theories of arrays

Decision procedures for various theories of arrays have been studied previously. Most of these theories can be divided into extensional and non-extensional varieties. In this section, several families of array theories are axiomatized in classical first-order multi-sorted logic with equality. The theory **Arr** decided in this chapter is then presented and compared to previously decided theories.

3.1.1 The language

Sorts The language has a basic sort I for indices into arrays. It also has value sorts, which are the sorts of individuals that may be stored in arrays. The sort V is the sort for primitive values stored in arrays. The set of value sorts is defined to be the least set X satisfying

- $V \in X$
- $\tau \in X \to \mathit{array}_\tau \in X$

Every value sort except V is an array sort. The value sorts together with I are all the sorts of the language. V and I need not be distinct.

Definition 3.1 (dimensionality of a value sort) *The dimension $\mathit{dim}(\tau)$ of a value sort τ is defined by*

- $\mathit{dim}(V) = 0$
- $\mathit{dim}(\mathit{array}_\tau) = \mathit{dim}(\tau) + 1$

Terms The language has countably infinitely many variables and constants of each distinct sort. The constants are uninterpreted, in the sense they will not occur in any axiom or axiom scheme. The function symbols of the language are

- read_τ of type ($\mathit{array}_\tau \to I \to \tau$), for every value sort τ
- write_τ of type ($\mathit{array}_\tau \to I \to \tau \to \mathit{array}_\tau$), for every value sort τ

Subscripts on *read* and *write* will generally be omitted. Informally, $\mathit{read}(a, i)$ will denote the value stored in array a at index i, and $\mathit{write}(a, i, v)$ will denote an array which stores the same value as a for every index except possibly i, where it stores value v.

Terms are built up in the usual way from constants and variables using the function symbols. Terms whose sort is an array sort will be called array terms. Terms whose sort is I will be called index terms. The dimension $\mathit{dim}(a)$ of an array term a is the dimension of its sort. If $\mathit{dim}(a) = n$, array a is said to be n-dimensional. If $n > 1$, then a is also said to be multi-dimensional.

Formulas The atomic formulas of the language are the equations between terms of the same sort. Formulas are built up from atomic formulas using propositional connectives and quantifiers in the usual way. A formula is *closed* if it has no free variables. A *literal* is an atomic formula or the negation of an atomic formula. A *theory* is a set of closed formulas.

3.1.2 Theories

Some theories restrict which array sorts are allowed. If a theory allows array sorts of dimension at most n, it is said to have just n-dimensional arrays. If a theory allows all array sorts, it is said to have multi-dimensional arrays.

Two axioms, both of which are schematic in a value sort τ, are used by different theories of arrays. The first is called the read-over-write axiom scheme. Informally, it says that for all arrays a, indices i and j, and values v of suitable type, reading the value stored at index j of $\mathit{write}(a, i, v)$ is v if the two indices are equal and $\mathit{read}(a, j)$ if they are different.

Axiom scheme 3.1 (read-over-write)

$$\begin{aligned}
&\forall\, a : \mathit{array}_\tau \,.\, \forall\, i : I \,.\, \forall\, j : I \,.\, \forall\, v : V \,. \\
&\qquad (i = j \rightarrow \mathit{read}(\mathit{write}(a, i, v), j) = v) \wedge \\
&\qquad (i \neq j \rightarrow \mathit{read}(\mathit{write}(a, i, v), j) = \mathit{read}(a, j))
\end{aligned}$$

Second is the extensionality axiom scheme. Informally, it expresses a principle of extensionality for arrays: if two arrays store the same value at index i, for each index i, they are equal.

Axiom scheme 3.2 (extensionality)

$$\begin{aligned}
&\forall\, a : \mathit{array}_\tau \,.\, \forall\, b : \mathit{array}_\tau \,. \\
&\qquad (\forall\, i : I \,.\, \mathit{read}(a, i) = \mathit{read}(b, i)) \rightarrow a = b
\end{aligned}$$

Extensional theories of arrays are those axiomatized by the read-over-write and extensionality axiom schemes. Non-extensional theories are axiomatized by just the read-over-write axiom scheme. Note that since a theory is a set of closed formulas, quantifier-free array theories have no variables; all 0-ary symbols are (uninterpreted) constants.

3.1.3 The theory Arr

The theory **Arr** decided in this chapter is the quantifier-free fragment of the extensional theory with multi-dimensional arrays where sort V is defined to be sort I. So indices are the values stored in 1-dimensional arrays.

The fully quantified extensional theory is undecidable, even in the absence of the function symbols write_τ and the read-over-write axiom scheme. This is because single-sorted first-order theories with function symbols and equality may be translated into this array theory in such a way that a first-order formula is valid iff its translation is. The translation maps constant symbols to index constants, n-ary function symbols to n-dimensional array constants, and terms like $f(i_1, \ldots, i_n)$ to nested read expressions

$$\mathit{read}(\ldots \mathit{read}(\mathit{read}(f', i'_1), i'_2) \ldots, i'_n)$$

where $f', i'_1, \ldots, i'_n$ are the translations of $f, i_1, \ldots, i_n$. The undecidability results for classical first order logic with just function symbols and equality (see, e.g., [E. 97]) can then be applied to show that even quite restricted quantified fragments of the extensional theory of arrays are undecidable.

A decision procedure for **Arr** may be useful even for applications which require a fully quantified logic. Many theorem provers, such as the widely used PVS [ORS92], provide strategies to reduce goals to subgoals in decidable fragments of their logic.

3.1.4 Comparison with related work

In this section, related work is summarized by describing which theories are decided. These theories often use axiomatizations different from but equivalent to that of **Arr**. All the theories decided are quantifier-free. Kaplan is the only one to distinguish the sorts V and I. Many of the previous theories allow arithmetic operators or uninterpreted functions over sort I to be used in addition to the symbols *read* and *write*. The restriction here to just the essential theory of arrays is justified by the fact that, as will be shown in Section 3.5 below, the satisfiability procedure for **Arr** is suitable for incorporation into CVC. The decision procedures for arithmetic and uninterpreted functions in CVC may thus be used together with the decision procedure for **Arr** to decide the combined theory.

The first two works present axioms but no decision procedure for their theories. With the exception of Levitt's work, the others give decision procedures for theories that are strictly weaker than **Arr**, either because they restrict the form of formulas in the theory (e.g., to just equations), disallow equations between arrays, or are non-extensional.

McCarthy In [McC62], McCarthy introduces the function symbols *read* and *write* and gives an informal semantics for an extensional theory of arrays based on them.

Collins and Syme Collins and Syme present in HOL a theory of finite higher-order partial functions similar to a theory with multi-dimensional arrays [CS95].

Kaplan In [Kap68], Kaplan gives a decision procedure for a non-extensional equational theory with just 1-dimensional arrays. He considers equations between index terms only, which is reasonable since his theory contains no non-trivial equations between arrays. He then shows how to extend his procedure to decide an extensional equational theory, where the equations may be between array as well as index terms. He imposes the restriction that distinct variables of sort I must receive distinct interpretations.

Suzuki and Jefferson In [SJ77], Suzuki and Jefferson present a decision procedure for a theory with just 1-dimensional arrays, where equations between arrays are not allowed. The theory has axioms for extensionality and the existence of constant arrays (arrays that store the same value at all indices), but these appear to be included for technical reasons only; the theory decided is equivalent to the one without those axioms under the restrictions they impose. They extend their procedure to decide a theory with a new predicate symbol *PERM*, where *PERM*(a, b) holds iff the multiset of the values stored in a is contained in the multiset of the values stored in b. Sentences of the theory are restricted to the form $P \rightarrow$ *PERM*(a, b), where P is any (quantifier-free) sentence not containing *PERM*. **Arr** does not have the *PERM* predicate, but inspection of the way Suzuki and Jefferson extend their algorithm to treat *PERM* shows that it could just as easily be used to extend the algorithm for **Arr**, as long as their restriction disallowing equations between array terms were retained.

Downey and Sethi In [DS78], Downey and Sethi present a decision procedure for an extensional equational theory with just 1-dimensional arrays. Equations between array terms are allowed. They prove that determining the invalidity of an equation in their theory of arrays is NP-complete.

Nelson and Oppen In [NO79], Nelson and Oppen describe an extensional theory of arrays. Their theory allows multi-dimensional arrays. They do not present their satisfiability procedure for the extensional theory, but in [Nel81], Nelson gives a detailed presentation of a satisfiability procedure for a non-extensional theory.

Levitt In Chapter 5 of his PhD thesis [Lev99], Levitt presents a decision procedure for an extensional theory of arrays based on solving equations and canonizing terms, in the style of Shostak [Sho84]. A detailed proof of correctness is not given, and has proved elusive to the authors. In contrast, a detailed proof of correctness is given below for the procedure for **Arr**.

3.2 The satisfiability procedure for Arr

Arr is decided by a refutation procedure. The procedure decides satisfiability of conjunctions of literals, which are equations and disequations between terms. Since propositional reasoning in CVC is handled by Chaff, as mentioned in Chapter 1, it suffices in describing a decision procedure of CVC to show how to decide satisfiability of a conjunction of literals. A conjunction of literals whose satisfiability is to be tested will be called a goal. Comma will be used to denote conjunction. Two goals are said to be *equisatisfiable* when one is satisfiable iff the other is.

3.2.1 Informal overview

The procedure works in two phases. In the first phase, the original goal is transformed into a set of subgoals such that (i) no subgoal contains *write* and (ii) the original goal is satisfiable iff one of the subgoals is. Eliminating write expressions is straightforward except when they occur as the left or right hand side of an equation. How to eliminate such occurrences of write expressions is the crucial insight of this algorithm.

Definition 3.2 ($=_{\_}$)

$$a =_{\mathcal{I}} b \quad \Leftrightarrow_{def} \quad \forall\, i : I \,.\, i \notin \mathcal{I} \rightarrow \mathit{read}(a, i) = \mathit{read}(b, i)$$

Formulas of the form $a =_{\mathcal{I}} b$ *with* $\mathcal{I} \neq \emptyset$ *are called partial equations.*

The crucial observation is that

$$\mathit{write}(a, i, v) = b \;\Leftrightarrow\; (a =_{\{i\}} b \wedge \mathit{read}(b, i) = v).$$

write expressions occurring as sides of equations may thus be eliminated by introducing partial equations.

The second phase of the procedure is based on the observation that in the absence of *write*, arrays behave like uninterpreted functions and *read* behaves like function application. So in the absence of *write*, a congruence closure algorithm (cf. [BT00]) could be used to decide the theory. The algorithm must be modified to work with partial equations as well as equations, but this can be done. For simplicity, a very simple congruence closure algorithm described is used, but it should be possible to modify a more complex algorithm.

3.2.2 Formal presentation

Figure 3.1 presents our procedure as a proof system. The proof system determines a non-deterministic procedure, where rules are applied bottom-up to analyze a goal into one or more subgoals. The system may be thought of as a rewrite system, where, for each rule, the goal below the line is rewritten to the subgoals above the line. The system resembles a Gentzen-Schütte system where only left rules of the corresponding sequent system are used (i.e., a sequent system where sequents are restricted to be of the form $\Gamma \Rightarrow \bot$). The derivable objects of this system are sets of literals. It is intended that a set of literals be derivable iff their conjunction is unsatisfiable. A *deduction* of a goal is a tree obtained by applying the proof rules bottom-up to that goal. Applying (ax) to a goal results in the trivial goal, which is the empty conjunction. The empty conjunction is identified with *FALSE*, and hence obviously unsatisfiable. Any non-trivial goal to which no rule can be applied is said to be *normal*. It is proved below that all normal goals are satisfiable.

The system has two phases. Some rules may be applied in just one phase, while others may be applied in either phase. The rules of phase 1 are applied to a goal until no rule applies, and then the rules of phase 2 are applied. The procedure stops and reports that the original conjunction is satisfiable if it encounters a normal subgoal. Otherwise, it reports that the original goal is unsatisfiable. As mentioned before, phase 2 is a modified

Phase 1:

(ext) $$\frac{\Gamma,\ \textit{read}(a,k) \neq \textit{read}(b,k)}{\Gamma,\ a \neq b}$$ k is not free in the conclusion; a and b are arrays

(r-over-w) $$\frac{\Gamma[v],\ i = j \qquad \Gamma[\textit{read}(a,j)],\ i \neq j}{\Gamma[\textit{read}(\textit{write}(a,i,v),j)]}$$

(w-elim) $$\frac{\Gamma,\ a =_{\mathcal{I}} b,\ i \in \mathcal{I} \qquad \Gamma,\ a =_{i,\mathcal{I}} b,\ \textit{read}(b,i) = v,\ i \notin \mathcal{I}}{\Gamma,\ \textit{write}(a,i,v) =_{\mathcal{I}} b}$$

(w-elim-helper) $$\frac{\Gamma,\ b =_{\mathcal{I}} a}{\Gamma,\ a =_{\mathcal{I}} b}$$ b is a write expression, and a is not

Phase 2:

(partial-eq) $$\frac{\Gamma,\ a =_{\mathcal{I}} b,\ \textit{read}(a,i) = \textit{read}(b,i),\ i \notin \mathcal{I} \qquad \Gamma,\ a =_{\mathcal{I}} b,\ i \in \mathcal{I}}{\Gamma,\ a =_{\mathcal{I}} b}$$
where $a \succeq b$; $\mathcal{I} \neq \emptyset$; $\textit{read}(a,i)$ occurs in Γ

(trans) $$\frac{\Gamma,\ a =_{\mathcal{I}} b,\ a =_{\mathcal{I}'} c,\ b =_{\mathcal{I} \cup \mathcal{I}'} c}{\Gamma,\ a =_{\mathcal{I}} b,\ a =_{\mathcal{I}'} c}$$ $\mathcal{I} \neq \emptyset$ and $\mathcal{I}' \neq \emptyset$

(subst) $$\frac{\Gamma[y],\ x = y}{\Gamma[x],\ x = y}$$ $x \succeq y,\ x \not\equiv y$, x not in $\Gamma[\,]$

(symm) $$\frac{\Gamma,\ y =_{\mathcal{I}} x}{\Gamma,\ x =_{\mathcal{I}} y}$$ $x \prec y$

Both phases:

($\in$-split) $$\frac{\Gamma,\ i = j \qquad \Gamma,\ i \in \mathcal{I}}{\Gamma,\ i \in (j, \mathcal{I})}$$

($\notin$-expand) $$\frac{\Gamma,\ i \notin \mathcal{I},\ i \neq j}{\Gamma,\ i \notin (j, \mathcal{I})}$$

($\in$-empty) $$\frac{}{\Gamma,\ i \in \emptyset}$$

(ax) $$\frac{}{\Gamma,\ x \neq x}$$

Figure 3.1: The decision procedure as a proof system

congruence closure algorithm. The core congruence closure algorithm consists of just the rules (symm) and (subst).

The set-theoretic operators have their usual meanings; note that $i, \mathcal{I}$ denotes $\{i\} \cup \mathcal{I}$, where $\mathcal{I}$ does not contain i. $\Gamma[\,]$ denotes a *context*, which is an expression containing one or more occurrences of a single free variable. The expression obtained by substituting the term t for the context's free variable is written $\Gamma[t]$. In the rule (subst), since the side condition requires that $\Gamma[\,]$ contain no occurrences of the term x, applying (subst) replaces all occurrences of x in $\Gamma[x]$ with the term y. $\equiv$ denotes syntactic identity. The symbol $\preceq$ denotes an ordering on terms by size, which is defined on terms in the usual way. Let $x \preceq y$ iff x and y are such that the size of x is less than or equal to the size of y. The variants $\prec$ and $\succeq$ are derived from $\preceq$ in the usual way.

3.2.3 Ensuring termination in phase 2

In phase 2, applications of (partial-eq) and (trans) must be restricted to ensure termination. There is nothing preventing (partial-eq) and (trans) from being applied repeatedly with the same partial equations, because for both rules, the partial equations are retained in the goal. For (partial-eq), this form of non-termination may be prevented by adding a side condition to the rule that prevents it from being applied if, informally, *read*(a, i) and *read*(b, i) are already known to be equal or if i is already known to be equal to an element of $\mathcal{I}$. Formally, the procedure can test whether or not t and t' are already known to be equal by applying all the rules of phase 2 except (partial-eq) and (trans) to the current goal with $t \neq t'$ added, and seeing whether or not that goal is reported unsatisfiable. If neither ($\in$-split) nor ($\notin$-expand) applies to the current goal, then this is equivalent just to comparing normal forms as determined by the core congruence closure algorithm. So in an implementation, this non-termination may easily be prevented. A similar approach can be used to prevent (trans) from being applied repeatedly to the same formulas. The required machinery, however, has been omitted from the proof system for simplicity.

3.3 Correctness of the Procedure

A satisfiability procedure is *sound* iff when it reports a goal unsatisfiable, the goal is indeed unsatisfiable. A procedure is *complete* iff when it reports a goal satisfiable, the goal is indeed satisfiable. A procedure is *correct* iff it terminates on all inputs, and it is sound and complete. In this section, a detailed proof of completeness for the satisfiability procedure for **Arr** is given. The proof of termination is routine. The following theorem implies soundness.

Theorem 3.1 (equisatisfiability) *The conclusion of each rule of the system is satisfiable iff one of its premises is satisfiable.*

Proof: The proof is routine. Consider just the rule (trans). If $a =_{\mathcal{I}} b$ and $a =_{\mathcal{I}'} c$ are true in some model, then it is easy to see by the definition of $=_{\_}$ that $b =_{\mathcal{I} \cup \mathcal{I}'} c$ is also true in some model. If c agrees with a at every index except those in $\mathcal{I}'$ and a agrees with b at every index except those in $\mathcal{I}$, then clearly $i \notin \mathcal{I} \cup \mathcal{I}'$ implies that c agrees with a at i and also that a agrees with b at i. Hence, c agrees with b at i. For the other direction, if the premise has a model, so does the conclusion, since the conclusion is a subset of the premise. □

Recall that a normal goal is a non-trivial goal (i.e., one which is not *FALSE*, the obviously unsatisfiable goal) to which no rule applies. By the equisatisfiability theorem, to prove completeness of the algorithm it suffices to show that any normal goal is satisfiable. This may be done by constructing a model for a normal goal. The following lemma is easily established.

Lemma 3.1 (effect of phase 1) *A goal that is normal with respect to phase 1 of the algorithm contains no write expressions and no disequations between array expressions.*

3.3.1 A convenient form for normal goals

In preparation for constructing a model, several transformations, which are not actually performed by the algorithm, are applied to a normal goal to give an equisatisfiable normal goal Γ, which is in a more convenient form. If the normal goal contains equations of the form $x = x$, clearly they may be removed and the result will be equisatisfiable. Next, modify the goal by doing the following. Let G be the goal as it currently stands. If there is a term of the form *read*(a, i) in G that is not the left hand side of any equation in G, choose a constant symbol c not occurring in G, and modify G by replacing *read*(a, i) everywhere in it with c and adding the equation *read*$(a, i) = c$ to it. If there is no such term *read*(a, i)

in G, stop. It is easy to show that the resulting goal is normal and equisatisfiable with the original normal goal. This resulting goal consists of formulas of one of the following four forms, where x, y, and z are constant symbols:

I. $\mathit{read}(x, y) = z$

II. $x \neq y$

III. $x =_{\mathcal{I}} y$, where every element of $\mathcal{I}$ is a constant symbol

IV. $x = y$

Since this resulting goal is normal, no formula $x = y$ of the form (IV) has its left hand side appearing anywhere else in the goal, since otherwise (subst) would apply. Let Γ be this resulting goal, except without the equations of the form (IV). Γ will be said to be in *convenient normal form*. Any model $\mathcal{M}$ of Γ may be extended to a model of Γ with those equations of the form (IV) by giving the same interpretation for the constant x as for the constant y, if $\mathcal{M}$ interprets y, and a single arbitrary interpretation for both x and y otherwise.

3.3.2 Construction of a model

In this section, a kind of term model for the goal Γ in convenient normal form is constructed. Several definitions, in terms of Γ, are required. The fact that the core congruence closure algorithm (rules (subst) and (symm)) is correct is used.

Definition 3.3 ($\rightarrow_{\_}$ and $\leftarrow_{\_}$) *Let $\rightarrow_{\_}$ and $\leftarrow_{\_}$ be the ternary relations defined, respectively, by*

$$a \rightarrow_{\mathcal{I}} b \quad \textit{iff} \quad (a =_{\mathcal{I}} b) \in \Gamma$$
$$a \leftarrow_{\mathcal{I}} b \quad \textit{iff} \quad (b =_{\mathcal{I}} a) \in \Gamma$$

Note that for any $\mathcal{I}$, the relations $\rightarrow_{\mathcal{I}}$ and $\rightarrow_{\mathcal{I}}$ need not be symmetric, since $(a =_{\mathcal{I}} b) \in \Gamma$ does not imply $(b =_{\mathcal{I}} a) \in \Gamma$.

Definition 3.4 ($\approx_{\_}$) *Let $\approx_{\_}$ be the least ternary relation satisfying*

1. $a \approx_{\emptyset} a$, *for every array constant* a *appearing in* Γ

2. $(a \rightarrow_{\mathcal{I}} b) \vee (b \rightarrow_{\mathcal{I}} a) \rightarrow a \approx_{\mathcal{I}} b$

Definition 3.5 ($\overset{*}{\approx}_{\_}$) *Let* $\overset{*}{\approx}_{\_}$ *be the least ternary relation containing* $\approx_{\_}$ *and satisfying*

$$(\exists\, c\,.\, a \approx_{\mathcal{I}} c \wedge c \overset{*}{\approx}_{\mathcal{I}'} b) \rightarrow a \overset{*}{\approx}_{\mathcal{I}\cup\mathcal{I}'} b$$

Definition 3.6 ($\overset{*}{\approx}$) *Let* $\overset{*}{\approx}$ *be the binary relation defined by*

$$a \overset{*}{\approx} b \quad \textit{iff} \quad \exists\, \mathcal{I}\,.\, a \overset{*}{\approx}_{\mathcal{I}} b$$

The context will help distinguish $\overset{*}{\approx}_{\_}$ and $\overset{*}{\approx}$. Note that $\overset{*}{\approx}$ is an equivalence relation.

Definition 3.7 (chains) *A chain of applications of a ternary symbol* R *like* $\approx_{\_}$ *or* $\rightarrow_{\_}$*, called an* R*-chain, is defined to be a conjunction of the form* $(a_1\ R_{\mathcal{I}_1}\ a_2) \wedge (a_2\ R_{\mathcal{I}_2}\ a_3) \wedge \ldots \wedge (a_{n-1}\ R_{\mathcal{I}_{n-1}}\ a_n)$*, with* $n \geq 2$.

- *The chain is denoted* $(a_1\ R_{\mathcal{I}_1}\ a_2\ R_{\mathcal{I}_2}\ \ldots\ R_{\mathcal{I}_{n-1}}\ a_n)$.
- n *is the* length *of the chain.*
- *The* union along the chain *is defined to be* $\bigcup_{1\leq j<n} \mathcal{I}_j$.
- *The chain is said to be* from x to y *iff* $a_1 \equiv x$ *and* $a_n \equiv y$.

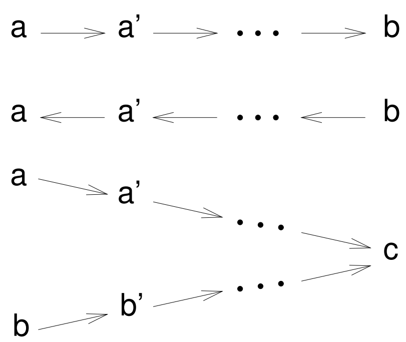

Figure 3.2: Standard forms for $\approx_{\_}$-chains

Lemma 3.2 (standard form for chains) *Suppose $a \stackrel{*}{\approx}_{\mathcal{I}} b$, with $\mathcal{I} \neq \emptyset$. Then one of the following is true:*

i. there is a $\rightarrow_{\_}$-chain from a to b or from b to a, where the union along the chain is $\mathcal{I}$

ii. for some c, there is a $\rightarrow_{\_}$-chain from a to c and another from b to c, where the union of the unions along the two chains is $\mathcal{I}$.

Figure 3.2 shows the possibilities.

Proof Let C be a $\approx_{\_}$-chain $a_1 \approx_{\mathcal{I}_1} \dots \approx_{\mathcal{I}_{n-1}} a_n$ from a to b, with $\mathcal{I} = \bigcup_{1 \le i \le n-1} \mathcal{I}_i$. Assume C is of minimal length of all such chains. For every i with $1 \le i \le n-1$, let $\leftrightarrow_i$ be either $\rightarrow_{\mathcal{I}_i}$ or $\leftarrow_{\mathcal{I}_i}$, and suppose we have $a_1 \leftrightarrow_i \dots \leftrightarrow_{n-1} a_n$. It is easy to prove that if this latter chain is not of one of the forms described in (i) and (ii), there must be an i with $1 < i \le n-1$ such that $\leftrightarrow_{i-1}$ is $\leftarrow_{\mathcal{I}_{i-1}}$ and $\leftrightarrow_i$ is $\rightarrow_{\mathcal{I}_i}$. So we have $a_{i-1} \leftarrow_{\mathcal{I}_{i-1}} a_i \rightarrow_{\mathcal{I}_i} a_{i+1}$. So both $a_i =_{\mathcal{I}_{i-1}} a_{i-1}$ and $a_i =_{\mathcal{I}_i} a_{i+1}$ are in Γ. It must be the case that both $\mathcal{I}_{i-1}$ and $\mathcal{I}_i$ are non-empty, since otherwise (subst) would apply to replace the left hand side of one of those equations by the right hand side of the other. No rules can apply, since Γ is normal. Since both $\mathcal{I}_{i-1}$ and $\mathcal{I}_i$ are non-empty, (trans) would be applicable, unless the conditions described in Section 3.2.3 for preventing non-termination were keeping it from being applied. This implies that either $a_{i-1} =_{\mathcal{I}_{i-1} \cup \mathcal{I}_i} a_{i+1}$ or $a_{i+1} =_{\mathcal{I}_{i-1} \cup \mathcal{I}_i} a_{i-1}$ is in Γ, since a_1 and a_2 must be their own normal forms as determined by the core congruence closure algorithm. Hence, we have $a_{i-1} \approx_{\mathcal{I}_{i-1} \cup \mathcal{I}_i} a_{i+1}$. So the chain $a_1 \approx_{\mathcal{I}_1} \dots a_{i-1} \approx_{\mathcal{I}_{i-1} \cup \mathcal{I}_i} a_{i+1} \dots \approx_{\mathcal{I}_{n-1}} a_n$, whose union is $\mathcal{I}$, has smaller length than C. This contradicts the assumption that C is of minimal length of such chains. $\Box$

Now an interpretation, given as a function $[\![\_]\!]$ from the constant and function symbols of Γ to their interpretations, is defined. $[\![\_]\!]$ is defined to map every constant symbol a of basic type I to a itself. $[\![\_]\!]$ will map array constants to functions. To satisfy extensionality, functions that give the same value for every input are required to be equal. First let $\perp_C$ be a new symbol not occurring in Γ, for every $\stackrel{*}{\approx}$-equivalence class C. Define $[\![read]\!]$ to be the operation of function application, except that when it is given $\perp_C$, it may just return $\perp_C$. Intuitively, for an array constant a, we will have $[\![a]\!]$ be a function mapping all but a finite number of inputs to a default value $\perp_C$. Formally, suppose a is in $\stackrel{*}{\approx}$-equivalence class C. Define $[\![a]\!]$ to be the function that returns $\perp_C$ for every input, except those assigned values by the following:

Definition 3.8 (interpretation of array constants) *We have:*
for every constant symbol b of the same type as a,
for every set $\mathcal{I}$ such that $a \overset{*}{\approx}_{\mathcal{I}} b$,
for every index constant i not appearing in $\mathcal{I}$,
if read$(b, i) = x \in \Gamma$ for some x, then
the value of $[\![a]\!]$ for input $[\![i]\!]$ is defined to be $[\![x]\!]$.

Notice that the body of Definition 3.8 may specify the value for $[\![a]\!]$ on input i more than once. So for $[\![\ ]\!]$ to be well-defined, if the value of $[\![a]\!]$ on input i is specified to be $[\![x_1]\!]$ and $[\![x_2]\!]$, we need $[\![x_1]\!] = [\![x_2]\!]$. So if $a \overset{*}{\approx}_{\mathcal{I}} b$ and $a \overset{*}{\approx}_{\mathcal{I}'} c$ with i not in $\mathcal{I}$ and not in $\mathcal{I}'$, then for $[\![\ ]\!]$ to be well-defined, it must be the case that if *read*$(b, i) = x_1$, *read*$(c, i) = x_2 \in \Gamma$, then $[\![x_1]\!] = [\![x_2]\!]$. Since the conditions $a \overset{*}{\approx}_{\mathcal{I}} b$, $a \overset{*}{\approx}_{\mathcal{I}'} c$, i not in $\mathcal{I}$, and i not in $\mathcal{I}'$ together imply $b \overset{*}{\approx}_{\mathcal{I} \cup \mathcal{I}'} c$ and i not in $\mathcal{I} \cup \mathcal{I}'$, the following lemma suffices to prove that $[\![\ ]\!]$ is indeed well-defined.

Lemma 3.3 (well-definedness of $[\![\ ]\!]$) *We have $x_1 \equiv x_2$ if the following conditions all hold*

- $a \overset{*}{\approx}_{\mathcal{I}} b$
- *i not in $\mathcal{I}$*
- *read*$(a, i) = x_1$, *read*$(b, i) = x_2 \in \Gamma$

The proof of this lemma relies on the following sublemma.

Lemma 3.4 (certain reads equal along chains) *Suppose $a_1, \ldots, a_n$, and i are such that $a_1 \rightarrow_{\mathcal{I}_1} \ldots \rightarrow_{\mathcal{I}_{n-1}} a_n$ for some $\mathcal{I}_1, \ldots, \mathcal{I}_{n-1}$, where i is not in $\bigcup_{1 \le j \le n-1} \mathcal{I}_j$. Suppose there is a constant x such that read$(a_1, i) = x \in \Gamma$. Then read$(a_n, i) = x \in \Gamma$.*

Proof The proof is by induction on n. The base case is trivial. For the induction case, suppose *read*$(a_1, i) = x \in \Gamma$. Since Γ is normal, no rules can apply. So we must have $\mathcal{I}_1 \neq \emptyset$, since otherwise (subst) would apply with $a_1 = a_2$ and *read*(a_1, i). Furthermore, since (partial-eq) cannot apply, it must be the case that the conditions of Section 3.2.3 for preventing non-termination are what is prohibiting its application with $a_1 =_{\mathcal{I}_1} a_2$ and *read*(a, i). In particular, it must be the case that *read*(a_2, i) is already known to be equal to *read*(a_1, i). The other possibility, namely that i is known to be equal to an element of

$\mathcal{I}$, is excluded because i is not in $\mathcal{I}$ by hypothesis, and correctness of the core congruence closure algorithm would require i to appear in $\mathcal{I}$ in a normal goal if i were known to be equal to an element of $\mathcal{I}$. For *read*(a_1, i) and *read*(a_2, i) to have the same normal form with respect to the core congruence closure algorithm, we must have *read*$(a_2, i) = x \in \Gamma$; this follows from the definition of convenient normal form. Now the induction hypothesis may be applied to conclude that *read*$(a_n, i) = x \in \Gamma$. □

Proof (of Lemma 3.3) Suppose $a \overset{*}{\approx}_{\mathcal{I}} b$ and suppose $\mathcal{I} \neq \emptyset$. Then by Lemma 3.2, there is either a $\rightarrow$_-chain from a to b or from b to a, or there is a constant c such that there is a $\rightarrow$_-chain from a to c and another from b to c. By Lemma 3.4, in the first case either *read*$(b, i) = x_1 \in \Gamma$ or *read*$(a, i) = x_2 \in \Gamma$, and in the second, *read*$(c, i) = x_1$, *read*$(c, i) = x_2 \in \Gamma$. Since Γ is normal, for all x, y, and z, we have that *read*$(x, i) = y$, *read*$(x, i) = z \in \Gamma$ implies $y \equiv z$, since otherwise (subst) would apply. So in either case, $x_1 \equiv x_2$. If $\mathcal{I} = \emptyset$, then it must be the case that $a \equiv b$, since *read*(a, i) and *read*(b, i) are both in Γ; otherwise, (subst) would apply. But again, *read*$(a, i) = x$, *read*$(a, i) = y \in \Gamma$ implies that $x \equiv y$. □

Lemma 3.5 (correctness of the constructed model) *The model constructed in the previous section satisfies every formula of the goal* Γ *in convenient normal form.*

Proof Consider the types (I), (II), and (III) of formulas from the list in section 3.3.1; recall that goals in convenient normal form consist of formulas of just these types.

Case I: *read*$(x, y) = z$ Since x is an array constant, $x \approx_{\emptyset} x$, and so the construction of Definition 3.8 will assign the value that function $[\![x]\!]$ takes on argument $[\![y]\!]$ to be $[\![z]\!]$. Hence $[\![$*read*$(x, y)]\!] = [\![z]\!]$.

Case II: $x \neq y$ Since all disequations in Γ are between index expressions, x and y must be index constants. Hence, $[\![x]\!] = x$ and $[\![y]\!] = y$, by construction. If $x \equiv y$, then the goal would not be normal, because (ax) would apply. So the interpretation satisfies $x \neq y$.

Case III: $x =_{\mathcal{I}} y$ It must be shown that for every index constant not in $[\![\mathcal{I}]\!]$, we have that $[\![x]\!]$ and $[\![y]\!]$ give the same value. But $[\![x]\!]$ and $[\![y]\!]$ have the same default value since they are in the same $\overset{*}{\approx}$-equivalence class. For those index constants i not in $\mathcal{I}$ that appear in a formula of the form *read*$(y, i) = z \in \Gamma$, they store the same values, by Definition 3.8. □

From the fact that a model has been constructed for a normal goal, the main result now follows.

Theorem 3.2 (completeness) *The satisfiability procedure for **Arr** is complete.*

3.4 Complexity analysis

Observe that each application of (w-elim) or (partial-eq) leads to one new subgoal for each element of the indexing set $\mathcal{I}$ in the rule. The size of $\mathcal{I}$ is easily seen to be bounded by the size N of the original goal Γ. So any deduction from Γ may be viewed as a tree with branching factor no more than N. It is not hard to show that N is an upper bound on the height of the tree, so there are at most $O(N^N) = O(2^{N \lg N})$ branches. Each branch can be shown to be of polynomial length, so the algorithm runs in worst-case exponential time.

Theorem 3.3 (NP-completeness) *The problem of testing a conjunction of literals for satisfiability in **Arr** is NP-complete.*

Proof Downey and Sethi showed that a subproblem of the problem decided here is NP-hard [DS78]. To show that the problem is in NP, observe that the size of the model constructed in the previous section for a goal Γ in convenient normal form is polynomial in the size of Γ. The conversion of a normal goal to convenient normal form incurs at most a polynomial expansion of the goal. So the size of the model constructed is polynomial in the size of the normal goal. Hence a model can be nondeterministically guessed in polynomial time. Checking whether or not a conjunction of literals is satisfied by a model can be done deterministically in polynomial time. So satisfiability of a conjunction of literals can be checked nondeterministically in polynomial time. $\Box$

3.5 Extensions

In this section, several extensions to the refutation procedure for **Arr** are considered.

3.5.1 Propagating all entailed equations

Full incorporation of the satisfiability procedure into CVC's framework for cooperating procedures [BDS00] requires that the procedure can discover all equations between terms occurring in a satisfiable goal that are entailed by that goal. The procedure for **Arr** always does this for index terms but not always for array terms. If the rules of Figure 3.3 are added to phase 2, however, it can be shown that if t and t' are array terms in a normal goal that are entailed to be equal, then $t \overset{*}{\approx}_{\emptyset} t'$.

$$\text{(trans2)} \quad \frac{\Gamma,\ a =_{\mathcal{I}} b,\ b =_{\mathcal{I}'} c,\ a =_{\mathcal{I} \cup \mathcal{I}'} c}{\Gamma,\ a =_{\mathcal{I}} b,\ b =_{\mathcal{I}'} c}$$
$$\text{where } \mathcal{I} \neq \emptyset \text{ and } \mathcal{I}' \neq \emptyset$$

$$\text{(patch)} \quad \frac{\Gamma,\ \neg\phi,\ a =_{i,\,\mathcal{I}} b \qquad \Gamma,\ \phi,\ a =_{\mathcal{I}} b}{\Gamma,\ a =_{i,\,\mathcal{I}} b}$$
$$\text{where } \phi \text{ is } \mathit{read}(a,i) = \mathit{read}(b,i)$$

Figure 3.3: Rules to propagate entailed equations

3.5.2 Propagating properly entailed disjunctions

Definition 3.9 (proper entailment of disjunctions) *A disjunction that is entailed when neither of its disjuncts is entailed is said to be* properly entailed.

Incorporating the procedure into CVC's framework also requires it to have the following property. Let ϕ and ψ be equations whose sides appear in goal Γ. If the procedure reports Γ satisfiable, then Γ cannot properly entail $\phi \vee \psi$. The original procedure for **Arr** does not have this property; an example is the normal goal

$$a =_{\{i\}} b, \quad a =_{\{j\}} b, \quad \mathit{read}(b,i) = v, \quad \mathit{read}(b,j) = v'$$

which entails $i = j \vee a = b$ but neither $i = j$ nor $a = b$. It can be proved, however, that the modified procedure of section 3.5.1 does have this property.

3.5.3 Allowing constant arrays

Constant arrays are arrays that store a single value for all indices. The language is extended with function symbols const_τ for each value sort τ, and the following axiom schema is added:

$$\forall\, x : \tau \,.\, \forall\, i : I \,.\, \mathit{read}(\mathit{const}(x), i) \quad = \quad x$$

The procedure of section 3.5.1 is modified to obtain a procedure for this extended theory by adding the rules of Figure 3.4. (const-elim1) is added to both phases, and (const-symm) and (const-elim2) are added to phase 2. In (const-elim2), the side condition states that the subscripting set on the partial equation is smaller than the set I of all indices. If I is infinite, this will always be true. If finite types are allowed (for example, subrange types like $[0..5]$), however, then there will be cases where the partial equation between two constant arrays says that they are equal everywhere except possibly at **all** indices. In that case, it would clearly be unsound to apply the rule to conclude that the values stored in the constant arrays are the same.

With this modified procedure, goals that are normal with respect to phase 2 may fail to be normal with respect to phase 1. For example, the applications of *const* in the goal $\mathit{const}(\mathit{write}(a, i, v)) = \mathit{const}(b)$ are removed using (const-elim2) in phase 2, but this adds the equation $\mathit{write}(a, i, v) = b$ to the goal, which could be analyzed with the (w-elim) rule of phase 1. So it is necessary to repeat the phases.

$$\text{(const-elim1)} \quad \frac{\Gamma[x]}{\Gamma[\mathit{read}(\mathit{const}(x), i)]}$$

$$\text{(const-symm)} \quad \frac{\Gamma,\ a =_{\mathcal{I}} \mathit{const}(x)}{\Gamma,\ \mathit{const}(x) =_{\mathcal{I}} a} \quad \text{where } a \text{ is not of the form } \mathit{const}(y)$$

$$\text{(const-elim2)} \quad \frac{\Gamma,\ x = y}{\Gamma,\ \mathit{const}(x) =_{\mathcal{I}} \mathit{const}(\tau, y)} \quad \text{if } |I| > |\mathcal{I}|$$

Figure 3.4: Rules to treat constant arrays

3.6 Conclusion

A refutation procedure for an extensional theory of multi-dimensional arrays has been presented and proved correct. The theory **Arr** decided essentially subsumes all previously decided array theories. The procedure is suitable for incorporation into CVC.

Chapter 4

Supporting Partial Functions

Partial functions occur frequently in Computer Science. Division is a partial function, since it is undefined when the divisor is zero. The function *car* which returns the head of a list is also partial, since it is undefined on the empty list. Reading the value at a given index in an array may be viewed as a partial function, since the result is undefined if the index is outside the bounds of the array. CVC must support partial functions like these to allow natural modeling of programs and hardware designs that use partial functions.

4.1 Approaches to partial functions

Unfortunately, there is no clear consensus in Computation Logic on the best way to handle partial functions. We include now a survey of some proposed approaches, which is organized by comparing the answers given for these two fundamental questions. What is the meaning of $f(x)$ when f is not defined on x? What should the meaning of an atomic formula like $t = t'$ be when one of the terms contains such a term $f(x)$ as a subterm?

4.1.1 Order-sorted algebra

In order-sorted algebra [GM92], function symbols f are declared with types $\sigma_1, \ldots, \sigma_n \to \tau$, where $\sigma_1, \ldots, \sigma_n, \tau$ are members of a set of basic sorts, which is partially ordered by a subsorting relation. If f is applied to arguments $a_1, \ldots, a_n$ where the sort of a_i is a subsort σ_i for all $1 \leq i \leq n$, then the expression $f(a_1, \ldots, a_n)$ is well-formed of sort τ. If, however, a_i has sort that is a supersort of σ_i for some $1 \leq i \leq n$, then a so-called retract must be inserted. A retract is a function r of type $\tau \to \tau'$, where τ is a maximal supersort of τ'. The following defining axiom for r is then added: $\forall\, a : \tau' .\ r(a) = a$. Inserting a retract turns $f(a_1, \ldots, a_n)$ into $f(a_1, \ldots, r(a_i), \ldots, a_n)$. Semantically, the value of the interpretation of r on arguments in the interpretation of τ but not in the interpretation of τ' is left unspecified. So the interpretation of a term like $\mathit{car}(r(\mathit{null}))$ (where r takes lists to non-null lists) is simply some unspecified value. This means that formulas like $\mathit{car}(r(\mathit{null})) = \mathit{car}(r(\mathit{null}))$ are valid. Order-sorted algebra is the basis for order-sorted rewriting, where retracts are inserted automatically when doing so makes some ill-sorted expression well-sorted. If an argument of some supersort can be rewritten to something that is of the required subsort, the retract can be removed. Otherwise, the presence of the unremoved retract in the expression serves as an error message.

Order-sorted algebra provides a foundation for an attractive and flexible form of equational reasoning in the presence of partial functions. But it is not obvious how to extend order-sorted algebra in a natural way beyond equational reasoning to general logical reasoning. Here is an example of the difficulty. Suppose in an informal mathematical discussion someone says, "Suppose the head of some list L is 3." Formally, this corresponds to introducing the hypothesis $\mathit{car}(L) = 3$. We could then reasonably expect it to be implied that L is of the form $\mathit{cons}(3, L')$ for some list L'. But with order-sorted algebra, a retract r will be inserted around the list L, so that the hypothesis is actually $\mathit{car}(r(L)) = 3$. There is a model M of this formula where L is the null list, and $r(L)$ is the interpretation of $\mathit{cons}(3, L')$ for some list L'. In this model M, then, L is the null list, so the hypothesis does not logically entail that $L = \mathit{cons}(3, L')$ for some L'.

The above example shows that what would appear to be the natural way to carry out general logical reasoning in order-sorted algebra does not capture a reasonable intuition about how partial functions should behave. As it stands, it is thus inadequate for use in CVC. This inadequacy is actually not surprising, because the conservativity result obtained in [GM92] seems only strong enough to support equational reasoning. The result says that adding retracts and their defining axioms preserves all entailed equalities between terms that do not contain retracts, and does not lead to the entailment of any new equalities. It is not clear how this can be extended beyond equalities, so that first-order formulas like $\mathit{car}(L) = 3 \ \to\ \exists\, L' .\, L = \mathit{cons}(3, L')$ are considered valid.

4.1.2 Logics with undefined terms

An alternative and older approach is that of logics with undefined terms. In these logics, the interpretation $[\![(f(x))]\!]$ of a term $f(x)$ may actually be undefined. An important point of difference between various logics of this kind is whether or not they are *bivalent*. A logic is bivalent if all formulas are either true or false in a given model (under a given variable assignment). In Kleene's three-valued logic and variants [Kle52, KK94, KK97], formulas may be either true, false, or undefined. Atomic formulas like $f(x) = f(x)$ are interpreted as undefined if the interpretation of $f(x)$ is undefined. A special unary function *!* may be added, applications $t!$ of which are interpreted as true if the interpretation of t is defined, and false otherwise.

This approach, while worthy of further study, seems ill-suited for many applications, since customary properties of classical logic fail. For example, the formula $p \rightarrow q$ is undefined if both p and q are. This means that $p \rightarrow p$, which is classically valid, is not valid in this logic, since if p is undefined, the formula is not true. This example also shows that the deduction theorem, which holds in classical logic, fails in this logic. The deduction theorem states that if $\Phi \cup \{\phi\} \models \psi$, then $\Phi \models \phi \rightarrow \psi$. We have that $p \models p$ in this logic, since any model which makes p true makes p true. But as just stated, $p \rightarrow p$ is not valid in this logic. The differences between this logic and classical logic in such fundamental properties as the set of propositional tautologies and basic meta-theorems like the deduction theorem is an unacceptably high price to pay for supporting partial functions.

An approach to undefined terms that does not suffer from these problems is adopted in the bivalent logics of [FG00] and [Fef95, Bee85]. In these logics, the interpretation of terms may be undefined as in Kleene's three-valued logic, but formulas are always interpreted as either true or false. The crucial idea that makes this possible is that the interpretation of an application $P(t_1, \ldots, t_n)$ of a predicate is false if the interpretation of any argument t_i is undefined. This has the effect of making formulas like the following valid:

$$
\begin{aligned}
0/x = 0/x \quad &\rightarrow \quad x \neq 0 \\
\mathit{car}(L) = 3 \quad &\rightarrow \quad \exists\, L' \,.\, L = \mathit{cons}(3, L')
\end{aligned}
$$

This is because the antecedents of those formulas are false if x is 0 and L is null, respectively. So if the antecedents are true, then $x \neq 0$ and $L \neq$ *null*. The consequents of the formulas then follow. Neither of those formulas is valid on the order-sorted or three-valued logics.

On this bivalent approach, the standard model-theoretic notion of a first-order structure is generalized to accomodate partial functions. Structures may interpret function symbols as partial operations on the carrier of the structure. These generalized structures we call *partial structures*. Partial structures where every function symbol is, in fact, interpreted as a total function we call *total structures*. The interpretations of the applications of function and predicate symbols are defined in the usual way when all arguments are defined. The semantics for the logical constants, including the quantifiers, is exactly the same as for classical first-order logic. This leads to the preservation of the set of classical tautologies and the deduction theorem.

We will work here only with functions that are *strict*, in the sense that the interpretation of an application $f(t_1, \ldots, t_n)$ of a function symbol is required to be undefined if the interpretation of any of the arguments t_i is undefined. The results that follow can probably be extended to non-strict functions, but this has not been worked out. Also, as in the three-valued approach, we may add a special unary predicate symbol $\downarrow$ to the language, applications of which are written in postfix notation. The definition of the semantics is extended so that $t\downarrow$ is interpreted as true iff the interpretation of t is defined.

4.2 Supporting partial functions in CVC

This section lays the groundwork for a general approach to supporting partial functions in CVC. The ultimate goal is to shift as much as possible of the burden of considering whether terms are defined or possibly undefined from the individual decision procedures onto the infrastructure of CVC. It would be highly desirable, for example, if decision procedures could provide suitable information to CVC's framework about which functions from its theory were partial, and then merely assume all terms it ever sees are defined. This ambitious goal has proved difficult to achieve. In this section we present a basic theorem relating validity in partial models to validity in total models. Using this result, we show how decision procedures for theories with partial functions can be integrated into CVC without changing CVC's underlying framework for cooperating decision procedures. The theories must satisfy certain restrictions, and the framework can only completely lift the burden from decision procedures of considering whether or not terms are defined in some very special situations. Nevertheless, even if further improvements to these results are desirable, they already enable CVC's infrastructure to provide useful support to decision procedures whose theories contain partial functions.

We prove an elimination theorem which shows how to test validity of a formula in partial models of a theory T by testing validity of a translation of the formula in total models. It is important to reduce the problem to validity checking in total models, since the results enabling decision procedures to be combined assume classical logic (and hence total models). We consider a theory T in the bivalent logic of partial functions. We divide the function symbols of T into two sets: those intended to denote total functions, and those intended to denote strictly partial functions. The former we call total function symbols, and the latter partial function symbols. For each n-ary function symbol f, we require the theory to have an axiom def_f, telling exactly when applications of f are defined. If f is a total function symbol, def_f simply states that f is total. If f is a partial function symbol, def_f states that f is defined on arguments $t_1, \ldots, t_n$ iff $\mathit{TCC}_f(t_1, \ldots, t_n)$ is true. We require TCC_f to be a quantifier-free formula not involving partial function symbols or $\downarrow$[1].

A translation of formulas from the bivalent logic of partial functions to classical logic is then defined which takes these *TCC* formulas into account. It then turns out that to check validity of a sentence in partial models of T, it is sufficient to check validity of the translation of the sentence in total models of T', where T' is just like T except that the def_f axioms have been removed, and any other axioms involving partial function symbols or $\downarrow$ have been translated. If there are no such axioms, then the original theory is carried over unmodified (except for the removal of the def_f axioms).

Elimination theorems that reduce validity in partial models to validity in total models have been proved for bivalent logics previously. In [FG00], an elimination theorem is proved which states that $T \models \phi$ with respect to partial structures iff $T^* \models \phi^*$ with respect to total structures, where the translation $(\_)^*$ is based on the following rewriting of atomic

[1]The name *TCC* is taken from PVS [ORS92], where the domain of definition of a function is described in that function's type. TCC stands for "type correctness condition".

formulas (extended homomorphically to all formulas):

$$P(t_1, \ldots, f(\bar{s}), \ldots, t_n) \;\longrightarrow\; (\exists\, y.\, P(t_1, \ldots, y, \ldots, t_n) \;\wedge\; P_f(\bar{s}, y)),$$

where P_f is a new predicate symbol corresponding to f.

This result cannot be readily used to support partial functions in CVC, for the following reasons. First, it is very natural to work with function symbols in most theories, and it would be inconvenient to eliminate them in favor of predicate symbols. Far worse, however, is the fact that the translation introduces quantifiers into formulas. Our decision procedures work only for quantifier-free formulas, so applying the translation maps formulas in our decidable fragment of first-order formulas outside that fragment. A final difficulty is that the theorem requires the entire background theory to be translated to support partial functions. We should attempt to avoid translating as much of the background theory as possible, since it may not be a trivial matter to modify the decision procedure to work with a translation of its theory.

Our elimination theorem is less general than that of [FG00], but it is better suited to supporting partial functions in CVC. The new theorem is based on a different $(\_)^*$ translation[2]. This translation replaces every atomic formula A in an arbitrary first-order formula ϕ with $(A \;\wedge\; A^*)$, where A^* is a formula, based on *TCC* formulas, which says that all terms in A are defined. For this to be useful in the case of CVC, applying the translation to a quantifier-free formula should yield a quantifier-free formula. This is obviously true if A^* is quantifier-free for all atomic formulas A. We place restrictions on the theory T to ensure that this is indeed the case.

4.2.1 Statement of the elimination theorem

Here is the statement of our elimination theorem. Several requirements are made of the background theory T which are currently needed to prove the elimination theorem. We think of the theory T as just a set of sentences; these are the axioms of the theory. We do not require T to be closed under implication. It is probably possible to relax some of the requirements on T, but that remains for future work.

[2]The crucial idea of this translation arose in conversations with Rick Sommer of Stanford's Center for the Study of Language and Information.

Theorem 4.1 (Elimination) *Let T be a theory in the bivalent logic of partial functions, such that the following hold:*

1. *The set of function symbols in T's language is divided into two disjoint sets: the total function symbols and the partial function symbols. For each n-ary total function symbol f in T's language, T contains a sentence def$_f$ of the form*

$$\forall\, x_1 \ldots \forall\, x_n \,.\, f(x_1, \ldots, x_n)\!\downarrow\; \leftrightarrow\;\; \mathit{TCC}_f(x_1, \ldots, x_n)$$

   If f is a total function symbol, TCC$_f$ is required to be TRUE. If f is a partial function symbol, TCC$_f$ is required to contain no quantifiers, partial function symbols, or $\downarrow$.

2. *Only a finite number of axioms of T contain a partial function symbol or $\downarrow$ (and hence, T contains only a finite number of partial function symbols, since sentences are finite)*

Then for any sentence ϕ in the language of T, the following are equivalent:

- $T \models \phi$ *in the bivalent logic of partial functions*
- $T' \models \phi^*$ *in classical logic*

where T' is the same as T except that

1. *T' does not contain the def$_f$ axioms of T*
2. *if α is any other axiom in T containing a partial function symbol or $\downarrow$, then T' contains α^* instead of α.*

Definition 4.1 (**($\_)^*$**) *The translation $(\_)^*$ is defined inductively by*

- $(P(t_1, \ldots, t_n))^* \;=\; (P(t_1, \ldots, t_n) \;\wedge\; t_1^* \;\wedge\; \ldots \;\wedge\; t_n^*)$
- $(g(t_1, \ldots, t_n))^* \;=\; (\mathit{TCC}_g(t_1, \ldots, t_n) \;\wedge\; t_1^* \;\wedge\; \ldots\; t_n^*)$
- $(x)^* \;=\;$ *TRUE, for variables x*
- $(\phi_1 \;\wedge\; \phi_2)^* \;=\; (\phi_1^* \;\wedge\; \phi_2^*)$

- $(\neg\phi_1)^* \;=\; \neg(\phi_1^*)$
- $(t{\downarrow})^* \;=\; t^*$
- $(\forall\, x.\, \phi)^* \;=\; \forall\, x.\, (\phi^*)$

This elimination theorem has the following consequences for adding support for partial functions to CVC. First, if all axioms α of theory T are such that α^* is classically equivalent to α, then the theory T' is simply identical to T except with the def_f axioms removed. This means that after applying $(\_)^*$ to the input formula as an initial prepocessing step, a decision procedure for T inside CVC may simply assume all terms it ever sees are defined.

For a practical example of this last result, consider the axiom for *car*, the function which selects the first element of a non-null list and is undefined on the null list. In the theory of inductive data types, we have a predicate *null?*, which is true only of the null list. So TCC_{car} would be simply $\neg\mathit{null?}(x)$. The defining axiom for *car* is

$$\mathit{car}(\mathit{cons}(n, L)) = n.$$

Notice that $(\_)^*$ of this axiom is classically equivalent to the axiom itself, since

$$\neg\mathit{null?}(\mathit{cons}(n, L))$$

is valid. So if we have a decision procedure which knows how to test validity in total models of this defining axiom for *car*, we can extend that decision procedure to one which will interpret *car* as a partial function. This is done just by pre-processing any input formula with $(\_)^*$ before giving it to the decision procedure for validity in total models. So we start with a decision procedure which considers *car* defined but unspecified on *null* and turned it into one which considers $\mathit{car}(\mathit{null})$ to be undefined. This is desirable, since if $\mathit{car}(\mathit{null})$ were defined but unspecified, then formulas like

$$\mathit{car}(L) = 3 \;\;\rightarrow\;\; \exists\, L'.\, L = \mathit{cons}(3, L')$$

would not be valid (for reasons similar to those in the discussion of order-sorted algebra in Section 4.1.1 above). But if $\mathit{car}(\mathit{null})$ is undefined, then such formulas are valid, since $\mathit{car}(\mathit{null}) = 3$ is interpreted as false.

This pleasant result is not quite the end of the story. The problem is that care must be taken in formulating the congruence axioms in the logic of partial functions to avoid making the theory inconsistent. For example, the following cannot consistently be used as the axiom of congruence for division:

$$(x_1 = y_1 \ \wedge \ x_2 = y_2) \ \Rightarrow \ (x_1/y_1 = x_2/y_2)$$

For consider the case where $x_1 = y_1 = 1$ and $x_2 = y_2 = 0$. Then the antecedent of the axiom is true. But its consequent is false according to our logic. This is because $1/0$ is undefined, and hence any atomic formula involving $1/0$, including even an equality like $1/0 = 1/0$, is false. The difficulty here is not a problem with our elimination theorem, which will correctly require that axiom to be translated to

$$(x_1 = y_1 \ \wedge \ x_2 = y_2) \ \Rightarrow \ (x_1/y_1 = x_2/y_2 \ \wedge \ y_1 \neq 0 \ \wedge \ y_2 \neq 0)$$

The problem is with the congruence axiom itself, which simply renders the theory inconsistent. The congruence axiom can be repaired as follows:

$$(x_1 = y_1 \ \wedge \ x_2 = y_2) \ \Rightarrow \ (x_1/y_1 \simeq x_2/y_2)$$

where $t \simeq t'$ is an abbreviation for

$$t_1 = t_2 \ \vee \ (\neg t_1{\downarrow} \ \wedge \neg t_2{\downarrow})$$

This $\simeq$ determines what is known as the *weak equality* relation. Or alternatively, the congruence axiom can be repaired in this way:

$$(x_1 = y_1 \ \wedge \ x_2 = y_2 \ \wedge \ y_1 \neq 0) \ \Rightarrow \ (x_1/y_1 = x_2/y_2)$$

Either modified axiom results in a consistent theory. It is easy to show, in fact, that the axioms are equivalent in the logic of partial functions (using the inference system of, say, [FG00]). If we replace $(x_1/y_1{\downarrow} \ \wedge x_2/y_2{\downarrow})$ with $(y_1 \neq 0 \ \wedge \ y_2 \neq 0$ in the expansion of $x_1/y_1 \simeq x_2/y_2$ in the first axiom, we can see that both axioms are classically equivalent to their translations under $(\_)^*$. The two axioms suggest two approaches for congruence reasoning inside decision procedures. Suppose the current logical context entails that $a = b$, and a decision procedure wishes to add to CVC's database of facts that, roughly, $t_1 = t_2$, where t_2 is the result of substituting a for b in t_1. In the presence of partial functions, it might be unsound to add $t_1 = t_2$, because t_1 could be undefined. But based on the first congruence axiom proposed above, it would be sound to conclude $t_1 \simeq t_2$. In an implementation, the decision procedure could actually assert $t_1 \simeq t_2$, and CVC's framework could translate this formula to

$$t_1 = t_2 \ \vee \ (\neg(t_1^*) \ \wedge \ \neg(t_2^*))$$

We can use t_1^* here instead of $t_1 \downarrow$ because we prove below (Lemma 4.4) that $(\_)^*$ is equivalent to $\downarrow$.

Alternatively, one could perform congruence reasoning in a decision procedure in a way suggested by the second proposed axiom above. When trying to conclude roughly that $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ when the ts equal the t's, the decision procedure could simply request that the framework ensure that the current logical context either entails $TCC_f(t_1, \ldots, t_n)$ or the negation of that formula. The framework has a mechanism for satisfying such requests by case-splitting. Then the decision procedure could just make sure this *TCC* formula holds before, say, adding the equation to the framework's database of facts.

One important note here is that CVC's framework performs classical reasoning. As noted above, the only difference at the level of logical reasoning between our bivalent logic of partial functions and classical logic is in the treatment of atomic formulas. Most of the basic services provided by the framework to subsidiary decision procedures do not actually perform logical reasoning. Of the few that do, only one involves reasoning about atomic formulas. This is in the code for the *simplifier*, which is responsible for carrying out some simplifications of expressions with respect to the current logical context of CVC. The simplifier transforms equalities of the form $t = t$ to *TRUE*. This is sound classically, but as we have seen, it is not sound in our logic of partial functions, since t could be undefined (in which case $t = t$ is equivalent to *FALSE*).

Since our goal in handling partial functions is to reduce the problem to classical logic, our only difficulty with the simplifier is making sure that subsidiary decision procedures in CVC and others (like users of CVC) who may call it have an appropriate interface to it.

In the case of decision procedures in CVC, if they call the simplifier, they must not count on it to reason within the logic of partial terms. Hence, if a formula simplifies to *TRUE*, it is not necessarily true that that formula is true in the logic of partial functions. Only in classical logic is it guaranteed to be true. As long as decision procedures are reasoning classically, which again is our whole goal in this enterprise, no confusion will arise. To provide correct behavior to users of CVC who call the simplifier, it is sufficient just to apply $(\_)^*$ to any formula that the user wishes to simplify before giving it to the simplifier.

In the final analysis, to decide a theory T satisfying the requirements of the elimination theorem, we have the following. Suppose only the *def*$_f$ axioms of T contain $\downarrow$. Suppose further that all the axioms α of T other than the *def*$_f$ axioms which involve partial function symbols are such that α^* is classically equivalent to α. Then simply by providing the *TCC* formulas for its partial function symbols to CVC's framework, the decision procedure may then assume all terms it sees are defined. The one exception is if it generates a new term by substituting equals for equals. In that situation, a modified congruence axiom must be used to avoid unsoundly asserting equations that would be false in the logic of partial functions due to undefinedness of a term.

4.2.2 Proof of the elimination theorem

The proof of the elimination theorem relies on an elimination lemma. The lemma considers a special case of the situation described in the elimination theorem. This case supposes that the theory contains a partial function symbol f which is completely *uninterpreted* (i.e., not mentioned in any axiom) except for a single *def*$_f$ axiom stating where it is defined. Note that the restriction that f is uninterpreted prevents us from having even a congruence axiom for equality with respect to f.

The lemma reduces validity of a sentence ϕ in this theory to validity of ϕ^* in the same theory, except that f is now required to be interpreted as total by all models, and the axiom *def*$_f$ has been removed. Note that it is necessary to remove *def*$_f$ from the theory, since otherwise it may contradict the requirement that f now be total.

Assuming this lemma, the proof of the elimination theorem is simple. By hypothesis, there are, in addition to the *def*$_f$ axioms, only a finite number of axioms $\alpha_1, \ldots, \alpha_n$ in the theory that involve partial function symbols or $\downarrow$. Write the theory as $T, \alpha_1, \ldots, \alpha_n$. Then using the deduction theorem (which, as mentioned previously, holds for our bivalent logic of partial functions), those axioms can be conjoined and placed as an antecedent of the formula ϕ whose validity is to be tested:

$$\begin{array}{lll} & T, \alpha_1, \ldots, \alpha_n & \models \phi \\ \Leftrightarrow & T & \models (\alpha_1 \wedge \ldots \wedge \alpha_n) \rightarrow \phi \end{array}$$

Then a simple inductive argument shows that we can eliminate the partial function symbols one by one by applying the $(\_)^*$ translation. We eliminate a function symbol by applying the $(\_)^*$ translation based on (only) its *TCC*. The def_f axioms for other function symbols will be translated by applying this version of $(\_)^*$ to them. They will not change, since by hypothesis, the def_f axiom for f contains no partial function symbol other than f. This brings us to a point where we have applied a version of $(\_)^*$ for each partial function symbol, which can be easily shown to be equivalent to applying the $(\_)^*$ translation that takes the def_f axioms for all the partial function symbols into account. So letting $\hat{T}$ be the part of the original theory not containing partial function symbols or $\downarrow$, and letting D be the axioms stating that the total function symbols are total, we end up with

$$\begin{array}{lll} & \hat{T}, D & \models (\alpha_1^* \wedge \ldots \wedge \alpha_n^*) \rightarrow \phi^* \\ \Leftrightarrow & \hat{T}, D, \alpha_1^*, \ldots, \alpha_n^* & \models \phi^* \end{array}$$

where the entailment is with respect to models interpreting what were the partial function symbols as total. Clearly then we can drop the D axioms if we also require the models to interpret the total function symbols as total. This gives the conclusion of the theorem, since the resulting notion of validity is with respect to total models, which are the same as classical models.

It remains to state our elimination lemma precisely and prove it, for which task we need a few preliminary definitions.

4.2.3 Logical preliminaries

We first recall some basic definitions from first-order (unsorted) classical logic with equality. A signature is a set of symbols, each with an associated arity, which is either of the form $\iota, \ldots, \iota \rightarrow \iota$ (for function symbols) or $\iota, \ldots, \iota \rightarrow$ *boolean* (for predicate symbols).

We view ι as a special case of the former arity, so constant symbols are viewed as a special case of function symbols. One of the predicate symbols is required to be $=$ with its usual arity. If Σ is a signature, Σ-terms are built up in the usual inductive way from variables, constant symbols, and function symbols. Atomic Σ-formulas are then defined as usual to be applications of predicate symbols in Σ to the appropriate numbers of Σ-terms. Σ-formulas are built up as usual from atomic Σ-formulas, the boolean operations $\wedge$ (conjunction) and $\neg$ (negation), and universal quantification. If ϕ is a Σ-formula with free variables $x_1, \ldots, x_n$ (ordered by a fixed arbitrary total ordering on variables), $\phi(t_1, \ldots, t_n)$ denotes the Σ-sentence obtained by simultaneously substituting t_i for x_i, for all $1 \leq i \leq n$. If ϕ has no free variables, we let $\phi(t_1, \ldots, t_n)$ denote ϕ. Σ-sentences are Σ-formulas with no free variables. A Σ-theory is a set of Σ-sentences.

The meanings of terms and formulas in a partial Σ-structure are defined as for classical logic (see, e.g., [End72, Hod97]), except in the cases (described in Section 4.1.2) where the partiality of functions plays a role. Slightly non-standardly, we write $M[\![X]\!]$ for the interpretation of X in structure M. We write *dom*(M) for the carrier of structure M. The predicate symbol $=$ is required to be interpreted as the identity relation on the carrier of the structure. If T is a Σ-theory and ϕ is a Σ-sentence, then $T \models \phi$ iff every Σ-model of T is a model of ϕ. We recall a definition for the truth of a formula with free variables in a structure.

Definition 4.2 (truth of formulas) *Suppose M is a Σ-structure, $a_1, \ldots, a_n \in$ dom(M), and ϕ is a Σ-formula with free variables contained in $\{x_1, \ldots, x_n\}$ (ordered by our arbitrary fixed total ordering on variables). Let Σ' be the signature extended with new constant symbols $\{a_1, \ldots, a_n\}$ (we assume for simplicity that $a_1, \ldots, a_n$ can be used as new constant symbols). Let ϕ' be the Σ'-sentence which is the same as ϕ except that x_i, where it occurs free in ϕ, has been replaced by a_i, for all $1 \leq i \leq n$. Let M' be the structure that is the same as M except that it additionally interprets a_i as a_i, for all $1 \leq i \leq n$. Then we say that $M \models \phi[a_1, \ldots, a_n]$ iff $M' \models \phi'$.*

4.2.4 Statement of the elimination lemma

Let Σ be a first-order signature. Let $f \in \Sigma$ be a function symbol taking $n > 0$ arguments. Let T be a Σ-theory not containing f. Let TCC_f be a Σ-formula with n free variables, but not containing f or $\downarrow$. In order to force just the partial function symbol f while leaving the interpretation of all other partial function symbols g partial, we will temporarily take TCC_g to be simply $g(t_1, \ldots, t_m)\downarrow$ for all m-ary $g \in \Sigma \setminus \{f\}$. We recall the definition of our translation $(\_)^*$ on expressions:

- $(P(t_1, \ldots, t_n))^* \ = \ (P(t_1, \ldots, t_n) \ \wedge \ t_1^* \ \wedge \ \ldots \ \wedge \ t_n^*)$
- $(g(t_1, \ldots, t_n))^* \ = \ (TCC_g(t_1, \ldots, t_n) \ \wedge \ t_1^* \ \wedge \ \ldots \ t_n^*)$
- $(x)^* \ = \ \mathit{TRUE}$, for variables x
- $(\phi_1 \ \wedge \ \phi_2)^* \ = \ (\phi_1^* \ \wedge \ \phi_2^*)$
- $(\neg\phi_1)^* \ = \ \neg(\phi_1^*)$
- $(t{\downarrow})^* \ = \ t^*$
- $(\forall\, x.\, \phi)^* \ = \ \forall\, x.\, (\phi^*)$

Also, let def_f be the formula

$$\forall\, x_1 \ldots \forall\, x_n \,.\, f(x_1, \ldots, x_n){\downarrow} \ \leftrightarrow \ \ TCC_f(x_1, \ldots, x_n).$$

Finally, let $\models_f$ be defined as follows. Suppose ϕ is a Σ-sentence. Then $T \models_f \phi$ iff every partial model of T which interprets the function symbol f as a total function is a model of ϕ. Also, if M is a partial Σ-structure interpreting function symbol f as a total function, and if $M \models \phi$, we will sometimes also write $M \models_f \phi$. The subscript f on $\models$ just indicates that the partial structure M interprets f as total. The lemma is then the following.

Lemma 4.1 (elimination) *Let ϕ be a Σ-sentence. Then*

$$T, \mathit{def}_f \models \phi \ \ \Leftrightarrow \ \ T \models_f \phi^*,$$

Our approach to proving Lemma 4.1 is to prove the contrapositive of each direction of the bi-implication. So for the $\Rightarrow$ direction, we assume that there is a model of T, $\neg\phi^*$ which interprets f as total, and we show that there must then be a model of T, def_f, $\neg\phi$. Since it models def_f, the model may interpret f as partial. For the $\Leftarrow$ direction, we assume there is a model of the latter kind, and prove there must be a model of T, $\neg\phi^*$ which interprets f as total. In each case, when we are required to prove that a model exists, we actually construct it from the model that is assumed to exist. We construct a model of def_f out of a model that interprets f as a total function by forcing f to be undefined wherever def_f says it should be. We construct a model interpreting f as total out of a model of def_f by making f return an arbitrary value wherever def_f says it is undefined. More formally, we have the following definitions.

Definition 4.3 (M_{def}) *If M is a Σ-structure, then M_{def} is defined to be the Σ-structure that is exactly like M except that $M_{def}[\![f]\!]$ is undefined on all tuples $\bar{a} \in M$ where $M \models_f \neg TCC_f[\bar{a}]$.*

Definition 4.4 (M_a) *Similarly, if M is a Σ-structure and $a \in dom(M)$, then M_a is defined to be the Σ-structure that is exactly like M except that $M_a[\![f]\!]$ equals a on all tuples $\bar{a} \in M$ where $M \models_f \neg TCC_f[\bar{a}]$.*

With these definitions, we can formulate the following lemma, using which the formal proof of Lemma 4.1 is easy:

Lemma 4.2 (model construction) *Let ϕ be a Σ-formula (possibly containing f, possibly containing $\downarrow$) with free variables $x_1, \ldots, x_n$. Let M be a Σ-structure, and let $a, a_1, \ldots, a_n$ be arbitrary elements of $dom(M)$. Then*

1. $M \models T, def_f, \phi[a_1, \ldots, a_n] \quad \Rightarrow \quad M_a \models_f T, \phi^*[a_1, \ldots, a_n]$
2. $M \models_f T, \phi^*[a_1, \ldots, a_n] \quad \Rightarrow \quad M_{def} \models T, def_f, \phi[a_1, \ldots, a_n]$

From this model construction lemma, we can then prove Lemma 4.1, our elimination lemma.

Elimination lemma: $\quad T, def_f \models \phi \quad \Leftrightarrow \quad T \models_f \phi^*$

Proof of elimination lemma ($\Leftarrow$): We prove the $\Leftarrow$ direction by proving its contrapositive. Suppose $T, def_f \not\models \phi$. Then there is a Σ-model M of T, def_f in which $\neg\phi$ holds. So by part (i) of Lemma 4.2, $M_a \models_f T, (\neg\phi)^*$. Since $(\neg\phi)^* \equiv \neg(\phi^*)$ by definition, we have $M_a \models_f T, \neg(\phi^*)$, which implies that $T \not\models_f \phi^*$, as required.

Proof of elimination lemma ($\Rightarrow$): We again prove the contrapositive. Suppose $T \not\models_f \phi^*$. Then there is a Σ-model M of T which interprets f as total and in which $\neg(\phi^*)$ holds. By part (ii) of Lemma 4.2, $M_{def} \models T, def_f, \neg\phi$. This implies that $T, def_f \not\models \phi$. $\Box$

4.2.5 Proof of part (i) of the model construction lemma

We state several helping lemma (proved in Section 4.2.7) needed for the proof. Then we prove part (i) of the model construction lemma. We note first the obvious fact that if $M \models \mathit{def}_f$, then M_a interprets f as a total function. This is because by the semantics, if $M \models \mathit{def}_f$, then $M[\![f]\!]$ is defined on all elements of the domain satisfying TCC_f. By construction, $M_a[\![f]\!]$ is the same as $M[\![f]\!]$ except that it is defined on all elements not satisfying TCC_f. Hence, $M[\![f]\!]$ is total.

In the following Lemmas 4.3 through 4.7, let Σ' be any signature extending Σ with new constant symbols. Let M be any Σ'-structure. Let t be any Σ'-term.

Lemma 4.3 *Suppose ϕ is a Σ'-sentence not containing f or $\downarrow$. Then*

1. $M \models \mathit{def}_f, \phi \quad \Rightarrow \quad M_a \models_f \phi$, *and*
2. $M \models_f \phi \quad \Rightarrow \quad M_{\mathit{def}} \models \phi$.

Lemma 4.4 (equivalence of $\downarrow$ and $(\_)^*$) $M \models \mathit{def}_f \quad \Rightarrow \quad (M \models t\!\downarrow \; \Leftrightarrow \; M \models t^*)$.

Lemma 4.5 (preservation of $(\_)^*$ in M_a) $M \models \mathit{def}_f \quad \Rightarrow \quad (M \models t^* \quad \Leftrightarrow \quad M_a \models_f t^*)$

Lemma 4.6 (preservation of $(\_)^*$ in M_{def}) $M \models t^* \quad \Leftrightarrow \quad M_{\mathit{def}} \models t^*$

Lemma 4.7 (value preservation) $M_{\mathit{def}} \models t\!\downarrow \; \Rightarrow \quad M[\![t]\!] = M_{\mathit{def}}[\![t]\!]$

We can now prove part (i) of **Lemma 4.2**, which says

$$M \models T, \mathit{def}_f, \phi[a_1, \ldots, a_n] \quad \Rightarrow \quad M_a \models_f T, \phi^*[a_1, \ldots, a_n]$$

Proof: Let us consider M to be its extension to a $(\Sigma \cup \{a_1, \ldots, a_n\})$-structure which interprets a_i as a_i for all $1 \leq i \leq n$. The proof proceeds by induction on the structure of ϕ. In each case, by Lemma 4.3, $M_a \models_f T$, since T does not contain f or $\downarrow$ by hypothesis. So we show only that $M_a \models_f \phi^*$.

Case $\phi \equiv P(t_1, \ldots, t_n)$: Assume $M \models T, \mathit{def}_f, P(t_1, \ldots, t_n)$. By the semantics, $M \models (t_1\!\downarrow \; \wedge \; \ldots \; \wedge \; t_n\!\downarrow)$. So by Lemma 4.4, $M \models (t_1^* \; \wedge \; \ldots \; \wedge \; t_n^*)$. By Lemma 4.5, this implies that $M_a \models_f (t_1^* \wedge \ldots \wedge t_n^*)$. We have that $M_a \models_f P(t_1, \ldots, t_n)$, using Lemma 4.7 and the fact that the interpretation of P is the same in M and M_a. So

$$M_a \models_f (P(t_1, \ldots, t_n) \wedge (t_1^* \wedge \ldots \wedge t_n^*)).$$

Hence $M_a \models_f P(t_1, \ldots, t_n)^*$ by definition of $(\_)^*$.

Case $\phi \equiv t{\downarrow}$: Assume $M \models T, \mathit{def}_f, t\downarrow$. So $M \models t^*$ by Lemma 4.4, and then $M_a \models_f t^*$ by Lemma 4.5. So $M_a \models_f (t{\downarrow})^*$, by definition of $(\_)^*$.

Case $\phi \equiv \phi_1 \wedge \phi_2$: Assume $M \models T, \mathit{def}_f, \phi_1 \wedge \phi_2$. By the induction hypothesis, $M_a \models_f \phi_1^*$ and $M_a \models_f \phi_2^*$, so $M_a \models_f (\phi_1 \wedge \phi_2)^*$, by definition of $(\_)^*$.

Case $\phi \equiv \forall\, x.\, \phi'$: Assume $M \models T, \mathit{def}_f, \forall\, x.\, \phi'$. By the semantics, this means that $M \models \phi'[a]$, for all $a \in \mathit{dom}(M)$. By the induction hypothesis, $M_a \models_f (\phi')^*[a]$. Since $\mathit{dom}(M) = \mathit{dom}(M_a)$, we have that $M_a \models_f \forall\, x.\, (\phi')^*$. So $M_a \models_f (\forall\, x.\, \phi')^*$, by definition of $(\_)^*$.

Case $\phi \equiv \neg\phi'$: We proceed by an inner induction on the structure of ϕ'.

Subcase $\phi' \equiv P(t_1, \ldots, t_n)$: Assume $M \models T, \mathit{def}_f, \neg P(t_1, \ldots, t_n)$. Suppose $M[\![t_i]\!]$ is defined, for all $1 \leq i \leq n$. Then $(M[\![t_1]\!], \ldots, M[\![t_n]\!]) \notin M[\![P]\!] = M_a[\![P]\!]$. So $M_a \models_f \neg P(t_1, \ldots, t_n)$. This implies that $M_a \models_f (\neg P(t_1, \ldots, t_n))^*$, since by the definition of $(\_)^*$,

$$(\neg P(t_1, \ldots, t_n))^* \equiv (\neg P(t_1, \ldots, t_n) \vee \neg t_1^* \vee \ldots \vee t_n^*).$$

Now suppose $M[\![t_i]\!]$ is undefined, for some $1 \leq i \leq n$. So $M \models \neg(t_i\downarrow)$, and by Lemmas 4.4 and 4.5, $M_a \models_f \neg(t_i^*)$. So again, $M_a \models_f (\neg P(t_1, \ldots, t_n))^*$.

Subcase $\phi' \equiv t{\downarrow}$: Assume $M \models T, \mathit{def}_f, \neg(t\downarrow)$. Then by Lemmas 4.4 and 4.5, $M_a \models_f \neg(t^*)$. So $M_a \models_f (\neg(t{\downarrow}))^*$, by the definition of $(\_)^*$.

Subcase $\phi' \equiv \phi_1 \wedge \phi_2$: Assume $M \models T, \mathit{def}_f, \neg\phi_1$ or $M \models T, \mathit{def}_f, \neg\phi_2$. By the inner induction hypothesis, $M_a \models_f (\neg\phi_1)^*$ or $M_a \models_f (\neg\phi_1)^*$. So $M_a \models_f (\neg(\phi_1 \wedge \phi_2))^*$.

Subcase $\phi' \equiv \forall\, x.\, \phi''$: Assume $M \models T, \mathit{def}_f, \exists\, x.\, \neg\phi''$. This means that there is $a \in \mathit{dom}(M)$ such that $M \models \neg\phi''[a]$. By the inner induction hypothesis, this implies that $M_a \models_f (\neg\phi'')^*[a]$. So $M_a \models_f (\exists\, x.\, \neg\phi'')^*$, by the semantics and the definition of $(\_)^*$.

Subcase $\phi' \equiv \neg\phi''$: Assume $M \models T, \mathit{def}_f, \neg\neg\phi''$. Then $M \models T, \mathit{def}_f, \phi''$, and by the outer induction hypothesis, $M \models_f (\phi'')^*$, and hence $M \models_f (\neg\neg\phi'')^*$. □

4.2.6 Proof of part (ii) of the model construction lemma

The proof of part (ii) of Lemma 4.2is similar to the proof of part(i). Part (ii) of **Lemma 4.2** says

$$M \models_f T, \phi^*[a_1, \ldots, a_n] \;\;\Rightarrow\;\; M_{\mathit{def}} \models T, \mathit{def}_f, \phi[a_1, \ldots, a_n]$$

Proof: As for the proof of part (i), we view M as its extension to a $(\Sigma \cup \{a_1, \ldots, a_n\})$-structure which interprets a_i as a_i for all $1 \leq i \leq n$. The proof proceeds by induction on the structure of ϕ. In each case, by Lemma 4.3, $M_{\mathit{def}} \models T$. By the semantics, we also have that $M_{\mathit{def}} \models \mathit{def}_f$, by definition of M_{def}. So we show only that $M_{\mathit{def}} \models \phi$.

Case $\phi \equiv P(t_1, \ldots, t_n)$: Assume $M \models_f T, (P(t_1, \ldots, t_n) \wedge t_1^* \wedge \ldots \wedge t_n^*)$. By Lemma 4.6, $M_{\mathit{def}} \models (t_1^* \wedge \ldots \wedge t_n^*)$. Then by Lemma 4.4, $M_{\mathit{def}} \models (t_1{\downarrow} \wedge \ldots \wedge t_n{\downarrow})$. So by Lemma 4.7, $M_{\mathit{def}}[\![t_i]\!] = M[\![t_i]\!]$, for all $1 \leq i \leq n$. Since $M \models_f P(t_1, \ldots, t_n)$, we have that $(M[\![t_1]\!], \ldots, M[\![t_n]\!]) \in M[\![P]\!]$. Since $M[\![P]\!]$ is the same as $M_{\mathit{def}}[\![P]\!]$, we then have that $(M_{\mathit{def}}[\![t_1]\!], \ldots, M_{\mathit{def}}[\![t_n]\!]) \in M_{\mathit{def}}[\![P]\!]$. Hence, $M_{\mathit{def}} \models P(t_1, \ldots, t_n)$.

Case $\phi \equiv t{\downarrow}$: Assume $M \models_f T, (t{\downarrow})^*$. Since $(t{\downarrow})^* \equiv t^*$, we can use Lemma 4.6 and then Lemma 4.4 to conclude that $M_{\mathit{def}} \models t{\downarrow}$.

Case $\phi \equiv \phi_1 \wedge \phi_2$: Assume $M \models_f T, (\phi_1 \wedge \phi_2)^*$. By the definition of $(\_)^*$ and then the induction hypothesis, we have that $M_{\mathit{def}} \models \phi_1$ and $M_{\mathit{def}} \models \phi_2$. So $M_{\mathit{def}} \models (\phi_1 \wedge \phi_2)$.

Case $\phi \equiv \forall x.\, \phi'$: Assume $M \models_f T, (\forall x.\, \phi')^*$. By the semantics and the definition of $(\_)^*$, this means that $M \models_f (\phi')^*[a]$, for all $a \in \mathit{dom}(M)$. By the induction hypothesis, $M_{\mathit{def}} \models \phi'[a]$. Since $\mathit{dom}(M) = \mathit{dom}(M_{\mathit{def}})$, $M_{\mathit{def}} \models \forall x.\, \phi'$.

Case $\phi \equiv \neg\phi'$: We proceed by an inner induction on the structure of ϕ'.

Subcase $\phi' \equiv P(t_1, \ldots, t_n)$: Assume that either $M \models_f T, \neg P(t_1, \ldots, t_n)$ or $M \models_f T, \neg t_i^*$, for some $1 \leq i \leq n$. If $M \models_f (t_1^* \wedge \ldots \wedge t_n^*)$ for all $1 \leq i \leq n$, then by Lemmas 4.6, 4.4, and 4.7, $M_{\mathit{def}}[\![t_i]\!]$ is defined and equals $M[\![t_i]\!]$, for all $1 \leq i \leq n$. So since $M_{\mathit{def}}[\![P]\!] = M[\![P]\!]$ and $(M[\![t_1]\!], \ldots, M[\![t_n]\!]) \notin M[\![P]\!]$, we have that $(M_{\mathit{def}}[\![t_1]\!], \ldots, M_{\mathit{def}}[\![t_n]\!]) \notin M_{\mathit{def}}[\![P]\!]$. So $M_{\mathit{def}} \models \neg P(t_1, \ldots, t_n)$.

If $M \models_f \neg t_i^*$ for some $1 \leq i \leq n$, then by Lemmas 4.6 and 4.4, $M_{\mathit{def}} \models \neg t_i{\downarrow}$. Then by the semantics, $M_{\mathit{def}} \models \neg P(t_1, \ldots, t_n)$, since the semantics requires that the interpretation of an application of a predicate to terms is false if the interpretation of any of the terms is undefined.

Subcase $\phi' \equiv t{\downarrow}$: Assume $M \models_f T, (\neg(t{\downarrow}))^*$. Then by Lemmas 4.6 and 4.4, $M_{\mathit{def}} \models \neg(t{\downarrow})$.

Subcase $\phi' \equiv \phi_1 \wedge \phi_2$: Assume $M \models_f T, (\neg\phi_1)^*$ or $M \models_f T, (\neg\phi_2)^*$. Then by the inner induction hypothesis, $M_{\mathit{def}} \models \neg\phi_1$ or $M_{\mathit{def}} \models \neg\phi_2$.

Subcase $\phi' \equiv \forall\, x.\, \phi''$: Assume $M \models_f T, (\exists\, x.\, \neg\phi'')^*$. By the semantics and the definition of $(\_)^*$, this means that there is $a \in \mathit{dom}(M)$ such that $M \models_f (\neg\phi'')^*[a]$. By the inner induction hypothesis, this implies that $M_{\mathit{def}} \models \neg\phi''[a]$. So by the semantics, $M_{\mathit{def}} \models \exists\, x.\, \neg\phi''$.

Subcase $\phi' \equiv \neg\phi''$: Assume $M \models_f T, (\neg\neg\phi'')^*$. So $M \models_f (\phi'')^*$. By the outer induction hypothesis, $M_{\mathit{def}} \models \phi''$. So $M_{\mathit{def}} \models \neg\neg\phi''$. $\Box$

4.2.7 Proof of Lemmas 4.3 through 4.7

We return now to prove our Lemmas 4.3 through 4.7. Recall that here, Σ' is any signature extending Σ with new constant symbols, M is any Σ'-structure, and t is any Σ'-term.

Lemma 4.3: Suppose ϕ is a Σ'-sentence not containing f or $\downarrow$. Then

1. $M \models \mathit{def}_f, \phi \;\Rightarrow\; M_a \models_f \phi$, and
2. $M \models_f \phi \;\Rightarrow\; M_{\mathit{def}} \models \phi$.

Proof: An easy induction on the structure of ϕ shows that the interpretation of f is irrelevant to the truth of ϕ in a structure, since ϕ does not contain f or $\downarrow$. By definition, M, M_a, and M_{def} are the same except for the interpretation of f. So if ϕ is true in M, then it is true in M_a and M_{def}. Furthermore, if $M \models \mathit{def}_f$, then M_a clearly interprets f as a total function, by the semantics. $\Box$

Lemma 4.4: $M \models \mathit{def}_f \;\Rightarrow\; (M \models t\!\downarrow \;\Leftrightarrow\; M \models t^*)$.

Proof: By induction on the structure of t. We prove the two directions of the bi-implication separately.

Case $(\Rightarrow)$ $t \equiv g(t_1, \ldots, t_n)$: Assume $M \models t\downarrow$. By the semantics, $M \models (t_1\downarrow \wedge \ldots \wedge t_n\downarrow)$. So by the induction hypothesis, $M \models (t_1^* \wedge \ldots \wedge t_n^*)$. If $g \equiv f$, then $M \models \mathit{TCC}_f(t_1, \ldots, t_n)$, since $M \models t\downarrow$ and $M \models \mathit{def}_f$. If $g \not\equiv f$, then we also have that $M \models \mathit{TCC}_g(t_1, \ldots, t_n)$, since in this case $\mathit{TCC}_g(t_1, \ldots, t_n) \equiv g(t_1, \ldots, t_n)\downarrow$, as we stipulated at the beginning of Section 4.2.4. So either way, $M \models (g(t_1, \ldots, t_n))^*$.

Case $(\Leftarrow)$ $t \equiv g(t_1, \ldots, t_n)$: Assume $M \models t^*$. By the definition of $(\_)^*$, we have that $M \models (\mathit{TCC}_g(t_1, \ldots, t_n) \wedge t_1^* \wedge \ldots \wedge t_n^*)$. If $g \not\equiv f$, then $\mathit{TCC}_g(t_1, \ldots, t_n) \equiv$

$g(t_1, \ldots, t_n)\downarrow$. So assume $g \equiv f$. By the induction hypothesis, $M \models (t_1\downarrow \wedge \ldots \wedge t_n\downarrow)$. So by the semantics and the fact that $M \models \mathit{def}_f$, we have that $M \models f(t_1, \ldots, t_n)\downarrow$. □

Lemma 4.5: $M \models \mathit{def}_f \;\Rightarrow\; (M \models t^* \;\Leftrightarrow\; M_a \models_f t^*)$

Proof: Assume $M \models \mathit{def}_f$. Then for the $\Rightarrow$ direction, an easy induction shows that t^* does not contain f (since TCC_f does not), so we may apply Lemma 4.3. For the $\Leftarrow$ direction, we may similarly apply Lemma 4.3 to show the contrapositive. □

Lemma 4.6: $M \models t^* \;\Leftrightarrow\; M_{\mathit{def}} \models t^*$

Proof: The proof is just like the one for Lemma 4.5. □

Lemma 4.7: $M_{\mathit{def}} \models t\downarrow \;\Rightarrow\; M[\![t]\!] = M_{\mathit{def}}[\![t]\!]$

Proof: The proof is by induction on the structure of t.

<u>Case $t \equiv g(t_1, \ldots, t_n)$:</u> Assume $M_{\mathit{def}} \models t\downarrow$. By the semantics, this implies that $M_{\mathit{def}} \models (t_1\downarrow \wedge \ldots \wedge t_n\downarrow)$. By the induction hypothesis, $M[\![t_i]\!] = M_{\mathit{def}}[\![t_i]\!]$, for all $1 \le i \le n$. Now $M[\![g]\!]$ agrees with $M_{\mathit{def}}[\![g]\!]$ on all tuples where $M_{\mathit{def}}[\![g]\!]$ is defined, by definition of M_{def}. Since $M_{\mathit{def}}[\![t]\!]$ is defined, we have

$$\begin{aligned}
M_{\mathit{def}}[\![t]\!] &= M_{\mathit{def}}[\![g]\!](M_{\mathit{def}}[\![t_1]\!], \ldots, M_{\mathit{def}}[\![t_n]\!]) \\
&= M[\![g]\!](M_{\mathit{def}}[\![t_1]\!], \ldots, M_{\mathit{def}}[\![t_n]\!]) \\
&= M[\![g]\!](M[\![t_1]\!], \ldots, M[\![t_n]\!]) \\
&= M[\![t]\!]
\end{aligned}$$

Chapter 5

Producing proofs from decision procedures

This chapter gives an overview of how proofs are produced from CVC, and describes in detail how proofs are produced from the database of equalities and the decision procedure for linear real arithmetic. The chapter includes a discussion of extensions to LF supported in the flea proof checker, which ships with CVC.

5.1 Overview of proof production

To produce a proof for a valid formula, the most straightforward approach is to instrument the code so that for every step of reasoning performed by the tool, the corresponding proof rule is applied. This simple idea has interesting consequences and refinements. First, there are many pieces of the system that perform tasks other than reasoning. Those pieces of the system will not require instrumentation. For example, parsing and printing of expressions does not require any modification for producing proofs (although, of course, new code may be required to print proofs in the correct format for input to the proof-checker). Furthermore, some parts of the system are concerned with reasoning, but do not actually perform a step of reasoning. For example, consider the proof rule for case-splitting. This rule says that a formula is unsatifiable iff it is unsatisfiable under the assumption of ϕ and also under the assumption $\neg\phi$. The formula ϕ is called a splitter. The rule does not require justification for why the particular splitter used was chosen, even though significant computation might have been required to choose it. So heuristics for splitter selection do not need to be instrumented.

There are two major pieces of the core decision procedure, each of which requires a different manipulation of proofs. The first major piece is the simplifier, which performs bottom-up rewriting of expressions. A decision procedure *DP* can be called from the simplifier to transform an expression E whose top-level function or predicate symbol is in *DP*'s theory. Suppose *DP* rewrites E to a new expression E'. Then it must produce a proof of the equation $E = E'$. For example, the decision procedure for inductive datatypes has code to perform rewrites like $\mathit{car}(\mathit{cons}(x, L)) \rightarrow x$. This code must produce a proof of $\mathit{car}(\mathit{cons}(x, L)) = x$. Simplification also requires a version of the Leibniz rule of substitutivity, which says that if $A_1 = B_1, \ldots, A_n = B_n$, then $f(A_1, \ldots, A_n) = f(B_1, \ldots, B_n)$. This rule must be used every time the bottom-up rewriting proceeds from subterms to terms immediately containing those subterms. Transitivity of equality must also be applied whenever a rewrite is applied to the result of another rewrite. At the end of a call to the simplifier where expression E is transformed to expression E', the proofs justifying the rewriting of subterms will have been stitched together using substitutivity and transitivity to produce a proof of $E = E'$.

The propositional solver incrementally asserts literals to the core decision procedure as it performs case splitting (see Chapter 2). CVC also allows the user to add literals statically to the current logical context before a query is performed. Added literals are first fully simplified, and then passed to the second major piece of the core decision procedure, which maintains a database of facts that are known to be true in the current logical context. For example, if the propositional solver has added the literals $A < B,\ B < C$ to the core, the database will contain those facts. The decision procedure for arithmetic will generate the fact $A < C$, which is also added to the database. For purposes of producing proofs, when a fact is added to the database, the proof of the fact must also be added, and the association between proof and fact must be maintained.

Literals that are dynamically added to the current logical context by the propositional solver are local hypotheses. They may be used by proofs only while they are active, and proofs require a notion of scope to prevent hypotheses that are no longer active from being used. In this simple example, $A < B$ and $B < C$ are local hypotheses. Their "proofs" are just the names of those hypotheses, say, *hyp*_37 and *hyp*_139. The proof of $A < C$ may then be represented as something like (*lt_trans hyp*_37 *hyp*_139), where *lt_trans* represents the proof rule for transitivity of $<$. *lt_trans* may be viewed as a function which for any X, Y, and Z, takes as input a proof of $X < Y$ and a proof of $Y < Z$ and returns a proof of $X < Z$.

5.2 The proof language

The language used for describing proofs supports exactly this functional view of proof rules. Proofs are represented using the Edinburgh Logical Framework (LF) [HHP93], a dependently typed lambda calculus developed to serve as a metalanguage for describing proof systems and other formal calculi. A proof system is encoded in LF as a set of type declarations. Proof checking is carried out by type checking, which is decidable in LF. While a complete introduction to LF and dependent type theory cannot be given here (see, for example, references like Section 9.5.2 of [Mit96], [Pfe01], Section 5 of [Bar92], or [Kam]), we consider a very simple example to demonstrate some of the basic ideas. The formal calculus for LF itself is presented in Chapter 6, which presents optimizations to the algorithm for type checking in LF. Section 5.4.4 discusses extensions to LF that are used to represent proofs from CVC more conveniently.

Consider the fragment of propositional logic with truth and conjunction as the only logical connectives. Let us use the proof rules given in Figure 5.1 for this fragment. Figure 5.2 gives an example of a proof in this system.

$$\text{(trueI)}\quad \frac{}{\vdash \textit{TRUE}}$$

$$\text{(andI)}\quad \frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}$$

$$\text{(andE1)}\quad \frac{\vdash A \wedge B}{\vdash A} \qquad\qquad \text{(andE2)}\quad \frac{\vdash A \wedge B}{\vdash B}$$

Figure 5.1: Proof rules for simple LF example

The encoding of this fragment of logic in LF proceeds as follows (we use the syntax for LF of Twelf [PS99]). We first declare an LF type for formulas:

$$\cfrac{\cfrac{\cfrac{}{\mathit{TRUE}} \quad \cfrac{}{\mathit{TRUE}}}{\mathit{TRUE} \wedge \mathit{TRUE}} \qquad \cfrac{}{\mathit{TRUE}}}{(\mathit{TRUE} \wedge \mathit{TRUE}) \wedge \mathit{TRUE}}$$

Figure 5.2: Example proof

```
o : type.
```

We must then represent the formula-building operations *TRUE* and $\wedge$:

```
TRUE : o.
AND : o -> o -> o.
```

These declarations say that *TRUE* is a formula and that *AND* takes two formulas and returns a formula. The type for *AND* is curried, since LF does not have tupletypes. The arrows associate to the right, so $o \to o \to o$ is to be parsed as $o \to (o \to o)$. The formula derived by our sample proof is then represented as

```
(AND (AND TRUE TRUE) TRUE),
```

which is clearly of type o, and hence intended to represent a formula. Note that application associates to the left, so that

$$(\mathit{AND}\ \mathit{TRUE}\ \mathit{TRUE})$$

is to be parsed as

$$((\mathit{AND}\ \mathit{TRUE})\ \mathit{TRUE}).$$

We may now represent the proof rules. The methodology used in representing proof systems in LF is to represent judgments like $\vdash P$ as types. This is in the spirit of the Curry-Howard isomorphism (see, e.g., Section 4.3.4 of [Mit96]). Since we have a family of judgments $\vdash P$, one for every formula P, this methodology requires the use of a family of types. The representation of $\vdash$ (we will use *pf* for its name) will be a type constructor, which takes in the representation of a formula and returns a type:

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

So if P is of LF type o, then $(\mathit{pf}\ P)$ is an LF type.

The encoding of the (trueI) proof rule is then very simple:

```
trueI : (pf TRUE).
```

This says that trueI is a proof of TRUE. The encoding of the (andI) rule is the first place in this example where dependent types are used:

```
andI : {A:o} {B:o} (pf A) -> (pf B) -> (pf (AND A B)).
```

Reading the type of andI from left to right, it says that andI is a function that takes in formulas A and B, a proof of A, and a proof of B; and returns a proof of the conjunction of A and B. Since $(\mathit{pf}\ P)$ is a type for any representation P of a formula, it makes sense to have functions with $(\mathit{pf}\ P)$ as domain or range. The functional view of proof rules mentioned at the start of this section is apparent here, since andI is represented as a function from proofs to proofs. The elimination rules for conjunction may then be represented as

```
andE1 : {A:o} {B:o} (pf (AND A B)) -> (pf A).
andE2 : {A:o} {B:o} (pf (AND A B)) -> (pf B).
```

Finally, our example proof is represented as

```
(andI (AND TRUE TRUE) TRUE
   (andI TRUE TRUE trueI trueI)
   trueI),
```

which has LF type

```
(pf (AND (AND TRUE TRUE) TRUE)).
```

5.3 Producing proofs from union-find

In the system described in [Nel81], and in CVC, a database of equalities is maintained using the union-find data structure (see, e.g., Section 22.3 of [CLR92]). This section describes how proofs are produced from this data structure.

5.3.1 Union-find

The union-find data structure maintains a set of equivalence classes of, in our case, expressions. Two operations are supported: merging the equivalence classes corresponding to two expressions (the "union" operation) and returning a canonical representative member of the equivalence class of an expression (the "find" operation). The representative is canonical in the sense that all members of the class have the same representative.

The union-find data structure maintains an attribute of each expression e, called e's *find pointer*. Initially, all expressions' find pointers are null. Given an expression e, the find operation follows find pointers from e until a null find pointer is encountered. It returns the expression with the null find pointer as the canonical representative for e's equivalence class. An important optimization used is path compression, where e's find pointer is made to point directly to the representative found by find, if the path from e to that representative is longer than 1. Given expressions e_1 and e_2 in different equivalence classes, the union operation sets the find pointer of e_1's representative (call it r_1) to point to e_2's representative (call it r_2), if e_1's equivalence class is smaller; otherwise, it sets r_2's find pointer to point to r_1. The size of each equivalence class C is maintained in an attribute of C's representative. A famously difficult analysis shows that for all m that may possibly arise in practical applications, a sequence of m unions and finds takes almost linear time [Tar75].

5.3.2 Producing proofs

We modify the union-find data structure so that it expects a proof of $e_1 = e_2$ for every operation *union*(e_1, e_2) and produces a proof that $e = r$ every time *find*(e) returns r. In addition to a find pointer, each expression e has an additional attribute which holds a proof. We maintain the invariant that the proof p stored in that attribute is a proof that $e = e'$, where e' is the expression pointed to by e's find pointer. We treat a null proof as a synonym for a proof by reflexivity of equality; this allows the invariant to hold initially, when all find pointers are null. The invariant must be maintained across the following two actions that change find pointers:

1. changing a find pointer to point directly to the representative during path compression

2. changing one representative's find pointer to point to another representative during a union

We show inductively how to maintain the invariant across (i), by showing how to maintain it over a single step of path compression. Suppose we have proofs p_1 and p_2 for find pointers from e_1 to e_2 and from e_2 to e_3 respectively. Then, as shown in Figure 5.3, when we change the pointer of e_1 to point to e_3, we change e_1's proof attribute to store (*eq_trans* e_1 e_2 e_3 p_1 p_2), where *eq_trans* is given the following LF type declaration (for simplicity, we use i here as the LF type for individuals):

```
eq_trans : {A:i} {B:i} {C:i}
           (pf (EQUALS A B)) -> (pf (EQUALS B C)) ->
           (pf (EQUALS A C)).
```

Now we show how to maintain the invariant across (ii). Here we need to use symmetry of equality, which is declared in LF as:

```
eq_symm : {A:i} {B:i} (pf (EQUALS A B)) -> (pf (EQUALS B A)).
```

Suppose union is called with e_1 and e_2, together with a proof p that $e_1 = e_2$. Suppose r_1 and r_2 are the representatives of the equivalence classes of e_1 and e_2, respectively. Let p_1 and p_2 be the proofs returned by the find operation that $e_1 = r_1$ and $e_2 = r_2$. If r_1's find pointer is modified to point to r_2, then r_1's proof attribute is set to

```
(eq_trans r1 e2 r2
   (eq_trans r1 e1 e2 (eq_symm e1 r1 p1) p)
   p2).
```

Let Q be this proof. Figure 5.4 shows how the attributes are updated. Dotted lines show proofs that do not correspond to a find pointer. Notice that there is a path from r_1 to r_2 using dotted and solid lines. This path determines the proof Q. Similarly, if r_2's find pointer is modified, then r_2's proof attribute may simply be set to (*eq_symm* r_2 r_1 Q). Alternatively, it may be set to this proof, which has the same number of new inferences as (*eq_symm* r_2 r_1 Q), but has depth shallower by one:

```
(eq_trans r2 e1 r1
   (eq_trans r2 e2 e1 (eq_symm e2 r2 p2) (eq_symm e1 e2 p))
   p1).
```

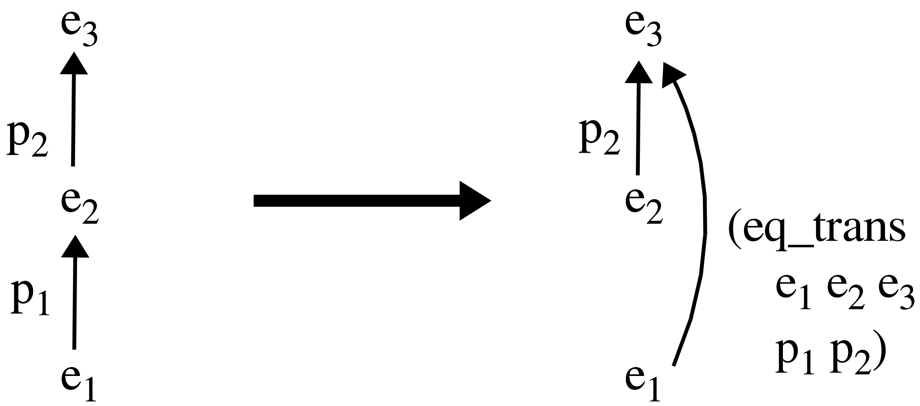

Figure 5.3: Maintaining proofs over one step of path compression

5.4 Producing proofs for linear real arithmetic

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.

As mentioned in previous chapters, propositional reasoning in CVC is separated from theory-specific reasoning. 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, since considering the canonizer and solver are sufficient to illustrate the main issues

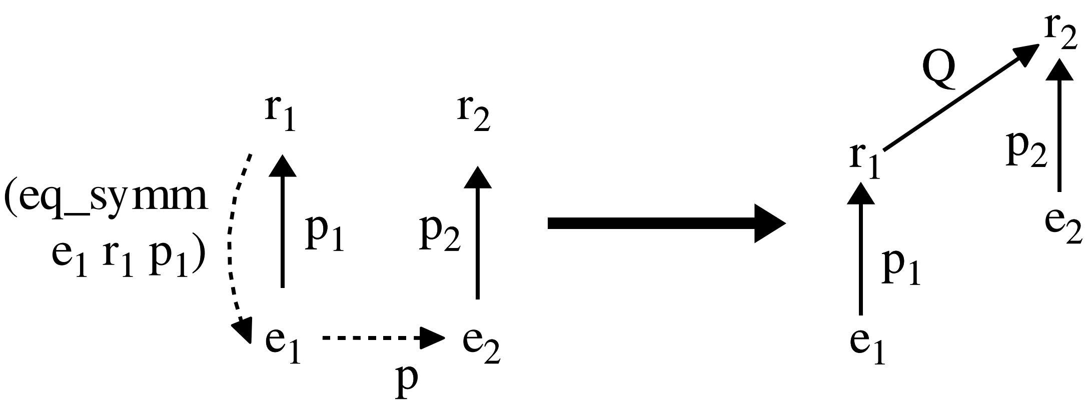

Figure 5.4: Maintaining proofs when a representative's find pointer changes during a union.

related to proof production.

5.4.1 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

1. $c, c_1, \ldots, c_n$ are constant arithmetic terms in coprime form
2. $x_1, \ldots, x_n \in \mathcal{V}$
3. for all $1 \le i < j \le n$, $x_i \prec x_j$

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.

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 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 5.4.1, 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.

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 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

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

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

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

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

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

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

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

Figure 5.5: Rewrites to flatten a linear arithmetic term

canonization, constant arithmetic terms like $2/4 + 3/6$ are put into coprime form. Finally, the trivial cases of flattened forms are rewritten:

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

5.4.2 Solver

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

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.

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.

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.

5.4.3 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 5.4.1), 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 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, selecting a monomial to isolate on one side of an equation is a constant-time operation in CVC, since accessing any monomial of a polynomial is a constant-time operation. But with an indirect representation of polynomials, isolating the i'th monomial first requires proving that that 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

$$(\textit{eq}\ (+\ (\textit{cons}\ 5\ (\textit{cons}\ (3 * x)\ (\textit{cons}\ (4 * y)\ \textit{null}))))\ 0),$$

then the most natural LF encoding for the isolation of the i'th monomial will require something like the following rule (using Twelf's syntax, where $\Pi\, x : A.B$ 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 $(\textit{remove\_ith}\ i\ L\ t\ L')$, 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 $(\textit{remove\_ith}\ i\ L\ t\ L')$. 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.

5.4.4 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

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

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:

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

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

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

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

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 [Jac99], 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*.

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 *nat* $\rightarrow$ *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.

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.

5.4.5 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 5.4. 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 LF_{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 $(\textit{pf}\, F)$:

```
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 $\{\{\textit{real}, (\textit{trm REAL})\}\}$ is the type used for these pairs (for the "{{ }}" notation, see Section 5.4.4 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 5.4.1 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]]))).
```

5.4.6 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 a conditional operator. $\mathrm{LF}_{\mathrm{nat}}$ 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 corresponding to valid formulas 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.

5.5 Alternative uses of proofs

As discussed in Chapter 1, proofs produced by a decision procedure have two primary uses. They can be checked to confirm results of the decision procedure, and they can be used to export results to other systems like proof assistants without requiring those other systems to trust the decision procedure. This section describes some other uses for proofs besides those two primary ones.

5.5.1 Abstract proofs

The proof of a valid formula represents a full trace of all logical reasoning performed to prove the formula. Such proofs may be quite large; on one benchmark, CVC produces in a few seconds a proof which is 2M of ascii text when represented as a directed acyclic graph (DAG), and over 1G when represented as a tree. In many cases, checking a proof takes significantly longer than it took CVC to produce it. So it is reasonable to consider abstracting away some of the information of a full trace to obtain an abstract proof, which may be much smaller than the full proof and may be checkable in a shorter period of time.

One example of a useful abstraction of proofs is to abstract away everything from the proof except the introduction and use of local hypotheses. Using this abstraction, the large proof just mentioned is reduced to 83k, which is well within the range that current proof-checkers can handle. An even more lossy version of this abstraction has been very useful. A hypothesis in a proof is *active* in that proof if its scope has not been closed in that proof. The second abstraction is to abstract away everything from the proof except the set of its active hypotheses. Abstracting a proof in this way reveals exactly which hypotheses are used by the full proof of the theorem. Any hypotheses that do not appear in the abstract proof are irrelevant. For example, suppose the user asserted $A < B$, the propositional solver asserted $C < D$, and the decision procedure then determined, by doing some case splits, that B must equal C and hence $A < D$. The active hypotheses in the proof of $A < D$ will be $A < B$ and $C < D$. Determining the set of hypotheses relevant to proving a given fact has several uses, which we turn to now.

5.5.2 Using proofs for testing and debugging

Given the complexity of CVC, debugging can be quite difficult. Many algorithms in CVC are intended to run in polynomial time in the size of a given formula, where the formula is represented as a DAG with maximal possible sharing of common subexpressions. So intermediate results must be cached, to avoid the exponential cost that may be incurred by traversing the formula as a tree. The use of stale cached results is a common source of bugs.

Abstracting away everything but the introduction and use of local hypotheses in a proof can be used to catch bugs where stale cached results are used. The use of a stale result manifests itself as the use of a local hypothesis outside its scope. A subtle caching bug in CVC's simplifier was discovered by generating this kind of abstract proof. The bug manifested itself in no other way, since the system did not crash, and the input formula, which was truly valid, was reported as valid. The bug was only detected when an error was reported checking the abstract proof. Because caching bugs like this one can be present in the system without any obvious sign of a problem, they are almost impossible to detect without proofs.

Another typical bug is an incompleteness bug. This is where the tool reports that the negation of the queried formula is satisfiable, but the set of formulas it reports as a context falsifying the formula is in reality unsatisfiable. Often the failure to detect the unsatisfiability of such a falsifying context is dependent on the state of CVC after it has executed for some time. If CVC is queried from scratch about whether or not the falsifying context by itself is unsatisfiable, in many cases it successfully reports that it is unsatisfiable. Frequently, only a small subset of the formulas in the falsifying context are relevant to its unsatisfiability. The abstract proof described above tells exactly what that subset is, which can suggest which part of the system is not performing correctly to lead to detection of the unsatisfiability.

Full and abstract proofs of various intermediate facts that the decision procedure has added to its database can be viewed while CVC is running. Even for large examples, the subset of active hypotheses relevant to the proof of an intermediate fact can be quite small. The implementors of the decision procedure can often reconstruct why the system added the new fact to its database, given the set of active hypotheses in the proof of that fact. This can make it easier to follow what the decision procedure is doing in a given execution. Unlike the full proof, the abstract proof can often be understood at a glance.

Proofs can be useful for determining code coverage of a test suite. Since proofs contain an application of the appropriate proof rule for every step of reasoning performed, the histogram of the uses of the various proof rules in the proofs of the test suite's formulas can serve as a profile of the test suite. Examining such a profile enables a tester to determine exactly which parts of the code that make steps of reasoning have been exercised. This coverage metric is similar to what is provided by a tool like gcov. Much more advanced metrics could be developed using proofs, because proofs contain information about execution paths. A tool like gcov merely allows the developer to determine which lines of code have been executed. The proofs show which paths through the code have been executed. Path-sensitive invariants of a particular execution could be determined from the proofs.

5.5.3 Using proofs for conflict clauses

As mentioned in Chapter 1, it is highly desirable for CVC to incorporate a state-of-the-art propositional SAT solver like Chaff [MSS99] for its propositional reasoning. Since CVC's logic is richer than just propositional logic, Chaff cannot be used directly. For full details on how Chaff is used in CVC, see [BDS02b]. The basic idea is to expose the propositional structure of an input formula to Chaff by replacing CVC's atomic formulas with new propositional variables, and maintaining an association between the new variables and the corresponding atomic formulas. Literals in Chaff are thus either propositional variables or their negations. When Chaff asserts that a literal is true, the corresponding atomic formula or its negation is added to the current logical context of the core decision procedure.

One of the most powerful techniques used in Chaff and other propositional solvers is the use of so-called conflict clauses. As the solver searches for a satisfying assignment to the propositional variables, it may discover that if certain literals $l_1, \ldots, l_n$ are all assigned to be true, then the formula cannot be satisfied. So for the formula to be satisfied, $\neg l_1 \vee \ldots \vee \neg l_n$ must be true. This disjunction is a conflict clause. The main benefit of conflict clauses comes from the fact that if the current assignment makes $n - 1$ of the disjuncts of the conflict clause false, the conflict clause implies that the remaining disjunct (call it l) must be true if the formula is to be satisfied. In this situation, the solver will actually assert the disjunct l. This can improve performance, because it could happen that asserting l also yields an assignment that cannot satisfy the formula. If l were not asserted, unnecessary case-splits on many other variables could be performed before l is chosen again as a splitter. All those other case-splits would be irrelevant to discovering that the current assignment to the $n - 1$ disjuncts of the conflict clause cannot satisfy the formula, regardless of the assignment to l.

When Chaff is used in CVC, a set of literals that appears satisfiable to Chaff may actually not be satisfiable, because of the interpretations the function and predicate symbols are given by the core decision procedure. So Chaff may assert a conjunction of literals, only to be told by the core decision procedure of CVC that that conjunction is unsatisfiable. In this case, Chaff could add the negation of the whole conjunction as a new conflict clause. But it can easily happen that all but a much smaller subset of the literals are irrelevant to the contradiction. For example, suppose Chaff asserts literals corresponding to $A < B,\ B < C,\ X = 0,\ Y = 0,\ Z = 0,\ C < A$, in that order. Only the first two literals and the last literal are relevant to the contradiction which the core decision procedure will discover. The abstract proof of the contradiction, which in this case is just $\{A < B,\ B < C,\ C < A\}$, tells exactly which assumptions are relevant to the contradiction. Since the conjunction of just the relevant assumptions is contradictory, Chaff needs to add only the negation of this potentially much smaller conjunction as a conflict clause.

Experiments show that this kind of pruning of conflict clauses from a decision procedure is crucial. Figure 5.6 shows the results on a 850MHz Pentium III. Each entry in the table gives the running time in seconds. Without pruning, the testcases ran much slower. Some other testcases (not shown here) finished in seconds with pruning but could not be completed at all before exhausting the 256M of main memory of the machine. The conclusion is that some form of pruning of the conflict clauses is required.

testcase	with no pruning	with pruning
pp-invariant	9.9s	0.2s
fb_6_12	8.5	0.1s
dlx-dmem	144s	1.8s

Figure 5.6: Run-times showing the benefits of pruning with abstract proofs

Chapter 6

Faster proof checking in LF

The ability for automated reasoning systems to produce easily verifiable proofs has been widely recognized as valuable (e.g., [Won99, BN00]). Recently, applications like proof-carrying code and proof-carrying authentication have created a new need for proofs that can be efficiently verified by a simple proof checker [Nec97, AF99]. The Edinburgh Logical Framework (LF) [HHP93] is 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 has the capability to produce proofs in a variant of LF for formulas it claims are valid. During validity checking, CVC computes values like a normal form n for a first-order term t. The computation produces a proof p that $t = n$. The normal form and the proof are both cached with t, and they can then both be reused later without repeating the computation. Internally, CVC represents proofs as a directed acyclic graph (DAG) with maximal sharing of common subexpressions. Reusing the cached normal form n for t results in a proof where the subproof p has more than one incoming edge in the DAG.

The problem then arises, how are proofs represented as DAGs to be checked? Since the DAG form for an expression can be exponentially smaller than the tree form, it would be disastrous to unfold the DAG to a tree by duplicating shared subexpressions wherever they occur. Proofs produced for large formulas by CVC can be megabytes long as a DAG, and at least gigabytes long as trees. Clearly, the proof checker must itself use some form of result caching if it is to check a proof represented as a DAG. The result that should be cached is the theorem (if any) that a subproof proves.

A naive approach to caching would have the proof checker cache the theorem which a subproof proves and reuse that result whenever that subproof is encountered subsequently. This approach is unsound, because syntactically identical subproofs may prove different theorems, depending on where they appear in a proof. This is because proofs introduce (named) local assumptions. For example, a proof

$$(\textit{modus-ponens}\ \textit{assumption}_1\ \textit{assumption}_2)$$

might prove B in a part of the proof where *assumption*$_1$ is an assumption that $A \rightarrow B$ holds and *assumption*$_2$ is an assumption that A holds; and it might prove C in a part of the proof where *assumption*$_1$ is an assumption that $A \rightarrow C$ holds. Proofs produced by CVC have this kind of dependency of subproofs on local assumptions. So some form of caching is required that respects the scope of local assumptions.

One approach that is sound but turns out to be inefficient is to maintain a table H mapping subproofs to the theorems they prove, and simply clear H during proof checking whenever the set of active assumptions changes. This approach is sound because multiple occurrences of the same subproof do indeed prove the same theorem under the same set of assumptions. It is inefficient because a subproof may depend on only a subset of the set of active assumptions, and so clearing all results may cause the theorem proved by that subproof to be computed again unnecessarily. A prototype proof checker using this approach was unable to check the proofs produced by CVC in a reasonable amount of time and space.

A natural alternative to clearing all cached results whenever an assumption **a** changes is to clear cached results only for those subproofs that depend on **a**. Also, there is no need to clear cached results eagerly, as soon as **a** changes. Results may be cleared lazily; whenever an attempt is made to reuse the theorem cached for some subproof, a check can be performed to see if the assumptions that the subproof depends on have changed. This is better than eager clearing of cached results, because not all subproofs will occur multiple times, and so not all cached results will need to be cleared. This lazy approach to caching has been implemented in a proof checker called flea, which is able to check example proofs produced by CVC in a modest amount of time and space that could not be checked at all previously.

The first contribution of this chapter is to formalize a generalization of this approach to context-dependent caching in the setting of LF, and prove it correct with respect to a standard algorithm for LF type-checking (Sections 6.1, 6.2, and 6.3). To improve the performance of type checking further, two other optimizations are developed: variable-sensitive safe substitution (Section 6.4) and incremental classifier computation for left-nested applications (Section 6.5). Experimental results using flea are given for proofs produced by CVC (Section 6.6). The chapter concludes with some observations about the relative performance of proof production from CVC and proof checking (Section 6.7). The chapter assumes some familiarity with type theory, but is essentially self-contained.

6.1 The Edinburgh Logical Framework

In this section, we briefly describe a standard calculus for LF. This is done in technical detail to enable the precise statement of subsequent theorems. Let *Sym* be an infinite set of symbols. A *binding* of a symbol $a \in$ *Sym* is an expression of the form $a : \tau$. If a list L of bindings contains a binding $x : \tau$, we say that L binds x. The addition of a binding $x : \tau$ to the end of a list of bindings Γ is denoted $\Gamma,\ x : \tau$.

Definition 6.1 (*lookup*) *If Γ is a list of bindings of symbols , then lookup(v, Γ) is defined inductively as follows. If Γ is of the form $\Gamma', x : \tau$, then lookup(v, Γ) equals τ if $v \equiv x$, and lookup(v, Γ') otherwise. If Γ is the empty list, then lookup(v, Γ) is undefined.*

Figure 6.1 presents a standard calculus for LF (cf. [Pfe01], Chapter 5 of [Bar92]). The primary derivable objects are sequents of the form $\Gamma \vdash X : Y$. It is required in all rules that $\vdash \Gamma$ *Ctx* is derivable. Symbols may be bound more than once by Γ in sequents; the *lookup* function is used to find the most recently added binding in Γ for a symbol. Bound variables may not be tacitly renamed. The notation $A[x := N]$ is used for the result of safely substituting expression N for symbol x in expression A. A formal definition of safe substitution is delayed till Section 6.4 below. The rule (app) requires the domain type A of the operator M and the type A' of the argument N to be equivalent ($A \cong A'$). This is different from other presentations, which usually require those types to be syntactically identical, not just equivalent; they then use a separate rule of conversion to re-classify N with type A, if indeed $A \cong A'$. The test used for equivalence is the term-directed, context-independent one of [Coq91], whose description is omitted here. It should be possible to extend the results to a context-dependent test like that of [HP00].

I. Classifications:

$$\text{(ax)} \quad \frac{}{\Gamma \vdash v : A} \quad A = \mathit{lookup}(v, \Gamma) \qquad\qquad \text{(type)} \quad \frac{}{\Gamma \vdash \mathit{type} : \mathit{kind}}$$

$$\text{(lam)} \quad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : \alpha}{\Gamma \vdash \lambda x : A.\, M : \Pi x : A.\, B} \quad \alpha \in \{\mathit{type}, \mathit{kind}\}$$

$$\text{(app)} \quad \frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A'}{\Gamma \vdash (M\ N) : B[x := N]} \quad A \cong A'$$

$$\text{(pi)} \quad \frac{\Gamma \vdash A : \mathit{type} \qquad \Gamma, x : A \vdash B : \alpha}{\Gamma \vdash \Pi x : A.\, B : \alpha} \quad \alpha \in \{\mathit{type}, \mathit{kind}\}$$

II. Contexts:

$$\text{(ctxemp)} \quad \frac{}{\vdash \cdot\ \mathit{Ctx}} \qquad\qquad \text{(ctxadd)} \quad \frac{\Gamma \vdash A : \alpha}{\vdash \Gamma, c : A\ \mathit{Ctx}} \quad \alpha \in \{\mathit{type}, \mathit{kind}\}$$

Figure 6.1: A standard presentation of LF

6.2 An annotation calculus for LF

This section presents a new calculus for LF, which is used to present the context-dependent caching optimization in Section 6.3 below. It is clear that some technical machinery beyond the standard calculus of Section 6.1 is required to describe caching. The technical device used here is that of typing annotations. The new calculus for LF derives an entire typing annotation of an expression e and all its subexpressions, as opposed to the type of a single expression only. Caching may then be formalized in a rule that says when typing information may be copied from one occurrence of a subterm of e to annotate another occurrence. An alternative technical device for formalizing caching, but one which we do not pursue here, might be some kind of explicit definition or abbreviation.

6.2.1 Preliminaries

The standard set-theoretic notion of partial function is assumed, together with related set-theoretic operations such as union, which will be performed on functions (cf. [FG00]).

For a unary partial function e, we use the notation $e(x) \downarrow$ to mean e is defined at x and $e(x) \uparrow$ to mean it is not. The domain of definition of e is denoted *Def*(e). Expressions are formalized as unary partial functions from the set of positions (Definition 6.2) to a set of symbols (as in, e.g., [CDG$^+$97]). This approach allows expressions and annotations of expressions to be treated uniformly. We assume a countably infinite set *Sym* of symbols (disjoint from $\{$*type*, *kind*, $\lambda, \Pi, @\}$.)

Definition 6.2 (*Pos* : the set of positions) *Let Nat be the set of natural numbers. The inductive set Pos of positions is defined by the following constructors:*

$$\epsilon : \mathit{Pos}$$
$$. : \mathit{Pos} \rightarrow \mathit{Nat} \rightarrow \mathit{Pos} \quad \text{(applications of “.” written infix)}$$

If $n \in$ Nat , n will often be used ambiguously to denote $\epsilon.n$.

We now define operations $\gg$ and $\ll$. The first right-shifts a position by prepending another position as a prefix, and the second left-shifts a position to remove a prefix. The notation is chosen to recall standard right and left bit-shifting operations. In meta-theoretic expressions, $\ll$ and $\gg$ will bind more loosely than the “.” constructor.

Definition 6.3 ($\ll$ and $\gg$: shifting positions) *If $\pi, \psi \in$ Pos and $n \in$ Nat , then the operation $\gg$ of prepending ψ to π is defined inductively by: $\epsilon \gg \psi = \psi$ and $\pi.n \gg \psi = (\pi \gg \psi).n$. Now suppose $\pi = (\pi' \gg \psi)$. Then $\pi \ll \psi$ is defined to be π'. Otherwise, it is undefined.*

Definition 6.4 (*Exp* : the set of LF expressions) *The set Exp of LF expressions is the set of all partial functions e from Pos to Sym $\cup \{$type, kind$, \lambda, \Pi, @\}$ satisfying the following requirements:*

- *e is undefined on all but a finite non-empty subset of Pos .*
- *Def(e) is prefix-closed: if $e(\pi.n) \downarrow$, then $e(\pi) \downarrow$.*
- *If $e(\pi) \in$ Sym $\cup \{$type, kind$\}$, then $\{i \mid e(\pi.i) \downarrow\} = \emptyset$.*
- *If $e(\pi) = @$, then $\{i \mid e(\pi.i) \downarrow\} = \{0, 1\}$.*
- *If $e(\pi) \in \{\lambda, \Pi\}$, then $e(\pi.0) \in$ Sym and $\{i \mid e(\pi.i) \downarrow\} = \{0, 1, 2\}$.*

We now extend $\gg$ and $\ll$ to expressions. The intention is that if e is an expression with $e(\pi)\downarrow$, then $(e \ll \pi)$ is the subexpression at position π in e (usually written $e|_\pi$).

Definition 6.5 ($\gg$ and $\ll$ extended) *The extensions of $\gg$ and $\ll$ which shift a function whose domain is Pos by a position are also denoted $\gg$ and $\ll$. For input $\pi' \in$ Pos, $(f \gg \pi)$ and $(f \ll \pi)$ are defined by*

$$\begin{array}{rcl}(f \gg \pi)(\pi') & = & f(\pi' \ll \pi)\\ (f \ll \pi)(\pi') & = & f(\pi' \gg \pi)\end{array}$$

Equivalently, viewing f as a set of pairs, $(\_ \gg \pi)$ and $(\_ \ll \pi)$ are applied to f by applying them to the first component of each pair. We also extend $\gg$ and $\ll$ to shift sets of positions by a position. The definition for $S \subseteq$ Pos is

$$\begin{array}{rcl}(S \gg \pi) & = & \{\psi \gg \pi | \psi \in S\}\\ (S \ll \pi) & = & \{\psi \ll \pi | \psi \in S\}\end{array}$$

Example 1 *Consider the following expression e:*

$$\{\epsilon \mapsto @,\ 0 \mapsto @,\ 0.0 \mapsto f,\ 0.1 \mapsto a,\ 1 \mapsto b\}$$

In customary notation (defined formally next), this is denoted $((f\ a)\ b)$, and $(e \ll 0)$ is $(f\ a)$, since, e.g., $(e \ll 0)(0) = e(0.0) = f$.

Notation 6.1 (expressions) *The customary notation for expressions is defined inductively, where we assume M_i denotes expression e_i for $i \in \{0, 1\}$:*

- *For any $s \in \{\mathit{type}, \mathit{kind}\} \cup \mathit{Sym}$, s denotes the function mapping ϵ to s and undefined everywhere else.*
- *$(M_0\ M_1)$ denotes $(e_0 \gg 0) \cup (e_1 \gg 1) \cup f$, where f maps ϵ to @ and is undefined everywhere else.*
- *for all $x \in$ Sym and $\Xi \in \{\lambda, \Pi\}$, $\Xi\, x : M_0.\, M_1$ denotes $(e_0 \gg 1) \cup (e_1 \gg 2) \cup f$, where f maps ϵ to Ξ, $\epsilon.0$ to x, and is undefined everywhere else.*

We claim without proof that all and only expressions are denoted by Notation 6.1. The usual structural induction on expressions is used in the sequel; we omit its justification to save space.

Definition 6.6 (annotations) *Suppose $e \in$Exp . An annotation of e is a partial function from Def(e) to Exp .*

An annotation of $e \in$*Exp* labels some subexpressions of e with other expressions.

6.2.2 A calculus of annotations

In this section, a new calculus for LF is presented. Derivable objects are essentially annotations of LF expressions. For a given $e \in$ *Exp* , the idea is to derive an annotation a of e such that for all $\pi \in$ *Def*(e), $a(\pi) = \tau$ iff $\Gamma \vdash e(\pi) : \tau$ is derivable in the standard calculus, where Γ is a context determined by the bindings occurring at prefixes of π in e. By deriving such annotations, we make all computed classifiers available at every point in the derivation. This sets the stage for reusing computed classifiers in Section 6.3. Similar technical machinery is used in [DV01]. There is also some resemblance to the marked calculus for LF of [Vir99].

Notation 6.2 (annotations) *Suppose a is an annotation of $e \in$Exp where*

$$\textit{Def}(a) = \{\pi_1, \ldots, \pi_n\}.$$

Then a may be denoted $\{\pi_1 : a(\pi_1), \ldots, \pi_n : a(\pi_n)\}$.

Figure 6.2 presents the calculus. The primary derivable objects are sequents $\Sigma \,|\, e \vdash a$, where a is an annotation and $e \in$*Exp* . It is required in all rules that $\vdash \Sigma$ *Sig* is derivable. The list Σ is called a signature instead of a context because it never changes during a derivation (after it has been proved well-formed using (sigemp) and (sigadd)). The (ax) rule specifies an expression's initial annotation, which is defined in the last part of Figure 6.2 by structural induction. The classificational rules (lam), (app), and (pi) just show how an annotation is extended to a new position. For the premises of those rules, the notation $\Sigma \,|\, e \vdash a \supseteq X$ means that $\Sigma \,|\, e \vdash a$ is derivable, where $a \supseteq X$.

A few further remarks on the calculus are needed. Meta-theoretic expressions like $(e \ll \pi.1)$ in the conclusion of (lam) denote the expression resulting from performing the shift. Also, α is either *type* or *kind*. The standard calculus is used computationally by applying the rules bottom-up to analyze a goal classification into subgoals. In contrast, the annotation calculus is used by applying the rules top-down to saturate the initial annotation specified by (ax). Finally, it is not obvious that the annotation calculus of Figure 6.2 is well-defined, since it is not obvious that the derived objects are always functional and not sometimes merely relational. Functionality up to equivalence will be a consequence of the equivalence of the annotation calculus with the standard presentation of LF, since LF enjoys unicity of classifiers up to equivalence (theorem 2.4 of [HHP93]).

Notation 6.3 (derivable sequents) *In addition to denoting sequents (theoretical objects), expressions like $\Sigma \,|\, e \,\vdash\, a$ will be used to denote the meta-theoretic proposition that the corresponding sequent is derivable in the annotation calculus.*

6.2.3 Equivalence with the standard calculus

This section proves the equivalence of the annotation calculus and the standard calculus of Section 6.1. We first state the theorem and three main lemmas used in its proof. The proof of the theorem and then the proofs of the lemmas follow.

Theorem 6.1 (correctness of annotation calculus) *Let $e, \tau \in$ Exp be arbitrary, and let Δ be an arbitrary list of bindings. Then the following are equivalent:*

1. *$\vdash \Delta$ Ctx and $\Delta \,\vdash\, e : \tau$ are both derivable in the standard calculus for LF.*
2. *There is an annotation a of e with $a(\epsilon) = \tau$ such that $\vdash \Delta$ Sig and $\Delta \,|\, e \,\vdash\, a$ are both derivable in the annotation calculus.*

Lemma 6.1 (monotonicity) *Suppose $\Sigma \,|\, e \,\vdash\, a$ is derivable starting from a sequent $\Sigma \,|\, e \,\vdash\, a'$ (used as an assumption to which the inference rules are then applied). Then $\Sigma \,|\, e \,\vdash\, a \cup X$ is derivable starting from sequent $\Sigma \,|\, e \,\vdash\, a' \cup X$. Furthermore, $\Sigma \,|\, e \,\vdash\, a$ implies $a \supseteq I_\Sigma(e)$.*

I. Classifications:

(ax) $$\frac{}{\Sigma \,|\, e \,\vdash\, I_\Sigma(e)}$$

(lam) $$\frac{\Sigma \,|\, e \,\vdash\, a \,\supseteq\, \{\pi.2 : \tau\} \quad \Sigma \,|\, \Pi\, x : \sigma.\, \tau \,\vdash\, a' \,\supseteq\, \{\epsilon : \alpha\}}{\Sigma \,|\, e \,\vdash\, a \cup \{\pi : \Pi\, x : \sigma.\, \tau\}} \qquad \begin{array}{l} e(\pi) = \lambda, \\ x = (e \ll \pi.0), \\ \sigma = (e \ll \pi.1) \end{array}$$

(app) $$\frac{\Sigma \,|\, e \,\vdash\, a \,\supseteq\, \{\pi.0 : \Pi\, x : \sigma.\, \tau,\;\; \pi.1 : \sigma'\}}{\Sigma \,|\, e \,\vdash\, a \cup \{\pi : \tau[x := (e \ll \pi.1)]\}} \qquad e(\pi) = @ \text{ and } \sigma \cong \sigma'$$

(pi) $$\frac{\Sigma \,|\, e \,\vdash\, a \,\supseteq\, \{\pi.1 : \mathit{type},\;\; \pi.2 : \alpha\}}{\Sigma \,|\, e \,\vdash\, a \cup \{\pi : \alpha\}} \qquad e(\pi) = \Pi$$

II. Signatures:

(sigemp) $$\frac{}{\vdash \cdot\, \mathit{Sig}}$$ (sigadd) $$\frac{\Sigma \,|\, e \,\vdash\, a}{\vdash \Sigma,\; c : e\; \mathit{Sig}} \qquad a(\epsilon) \in \{\mathit{type}, \mathit{kind}\}$$

III. Initial annotation:

$$\begin{array}{lll}
 & I_\Sigma(\mathit{type}) & = \{\epsilon : \mathit{kind}\} \\
 & I_\Sigma(\mathit{kind}) & = \emptyset \\
 & I_\Sigma(M_0\; M_1) & = (I_\Sigma(M_0) \gg 0) \,\cup\, (I_\Sigma(M_1) \gg 1) \\
\text{for } \Xi \in \{\lambda, \Pi\}, & I_\Sigma(\Xi\, x : \sigma.\, M) & = (I_\Sigma(\sigma) \gg 1) \,\cup\, (I_{\Sigma,\, x:\sigma}(M) \gg 2) \\
\text{for } v \in \mathit{Sym}, & I_\Sigma(v) & = \begin{cases} \{\epsilon : \mathit{lookup}(v, \Sigma)\} & \text{if } \mathit{lookup}(v, \Sigma) \downarrow, \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

Figure 6.2: Annotation calculus for LF

Proof: Both claims follow by induction on the assumed derivation: every rule only extends annotations (starting from the initial annotation $I_\Sigma(e)$), and no rule is prevented from being applied by definedness of an annotation at a position. $\Box$

Lemma 6.2 ($\gg$-shifting sequents) *Let* $\Xi \in \{\lambda, \Pi\}$ *and* $i \in \{0,1\}$ *be arbitrary.*

$$\begin{array}{llll} i. & \Sigma \,|\, M_i \vdash a & \text{implies} & \Sigma \,|\, (M_0\, M_1) \vdash \quad (a \gg i) \cup I_\Sigma(M_0\, M_1) \\ ii. & \Sigma \,|\, \sigma \vdash a & \text{implies} & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash \quad (a \gg 1) \cup I_\Sigma(\Xi\, x : \sigma.\, M) \\ iii. & \Sigma, x : \sigma \,|\, M \vdash a & \text{implies} & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash \quad (a \gg 2) \cup I_\Sigma(\Xi\, x : \sigma.\, M) \end{array}$$

Lemma 6.3 ($\ll$-shifting sequents) *Let* $\Xi \in \{\lambda, \Pi\}$ *be arbitrary.*

$$\begin{array}{llll} i. & \Sigma \,|\, (M_0\, M_1) \vdash a & \text{implies} & \Sigma \,|\, M_i \vdash \quad (a \ll i),\ \text{for } i \in \{0,1\} \\ ii. & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash a & \text{implies} & \Sigma \,|\, \sigma \vdash \quad (a \ll 1) \\ iii. & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash a & \text{implies} & \Sigma, x : \sigma \,|\, M \vdash \quad (a \ll 2) \end{array}$$

Proof (Theorem 6.1): Both directions ($(1) \Rightarrow (2)$ and $(1) \Leftarrow (2)$) of the claim are proved by induction on the structure of the assumed derivation (call it d) to show derivability of the desired sequent (call is S). The proof of the part of the claim about signatures and contexts is easy, as are the base cases for the classificational claims. The inductive cases for the classificational claims are proved as follows.

<u>Case (lam) ($\Rightarrow$):</u> d ends in

$$\text{(lam)} \quad \frac{\Delta, x : A \vdash M : B \qquad \Delta \vdash \Pi\, x : A.\, B : \alpha}{\Delta \vdash \lambda\, x : A.\, M : \Pi\, x : A.\, B} \quad \alpha \in \{\mathit{type}, \mathit{kind}\}$$

By the induction hypothesis, sequents $\Delta, x : A \,|\, M \vdash a'$ and $\Delta \,|\, \Pi\, x : A.\, B \vdash a''$ are derivable, with $a'(\epsilon) = B$ and $a''(\epsilon) \in \{\mathit{type}, \mathit{kind}\}$. The following sequent is then derivable by (iii) of Lemma 6.2 (shifting), applied to the first of these sequents:

$$\Delta \,|\, \lambda\, x : A.\, M \vdash (a' \gg 2) \cup I_\Delta(\lambda\, x : A.\, M)$$

Since $(a' \gg 2)(2) = a'(\epsilon) = \tau$, (lam) may be applied to this later sequent and the sequent $\Delta \,|\, \Pi\, x : A.\, B \vdash a''$ to conclude the desired sequent S.

Case (lam) ($\Leftarrow$): d ends in

$$\text{(lam)}\quad \frac{\Delta \,|\, \lambda x : \sigma.\, M \;\vdash\; a \;\supseteq\; \{2 : \tau\} \qquad \Delta \,|\, \Pi\, x : \sigma.\, \tau \;\vdash\; a' \;\supseteq\; \{\epsilon : \tau\}}{\Delta \,|\, \lambda x : \sigma.\, M \;\vdash\; a \cup \epsilon : \Pi\, x : \sigma.\, \tau}$$

Applying (iii) of Lemma 6.3 (shifting) to the first of these sequents and using the definition of $\ll$, the following sequent is then derivable:

$$\Delta, x : \sigma \,|\, M \;\vdash\; (a \ll 2) \;\supseteq\; \{\epsilon : \tau\}$$

Applying the induction hypothesis to this sequent and the sequent $\Delta \,|\, \Pi\, x : \sigma.\, \tau \;\vdash\; a'$ then gives the following sequents:

$$\Delta, x : \sigma \;\vdash\; M : \tau \qquad\qquad \Delta \;\vdash\; \Pi\, x : \sigma.\, \tau : \alpha$$

where $\alpha \in \{\mathit{type}, \mathit{kind}\}$. Now (lam) may be applied to derive the desired sequent S.

All other cases: Similar to the cases for (lam). Part (i) of the appropriate shifting lemma is used in each of the cases for (app). Part (ii) of the appropriate shifting lemma is used in each of the cases for (pi). $\Box$

We now prove Lemmas 6.2 and 6.3. Some preliminary lemmas about $\gg$ and $\ll$ are required.

Lemma 6.4 (appending using $\gg$) *For any $\pi \in \mathit{Pos}$ and $n \in \mathit{Nat}$, $\pi.n = n \gg \pi$.*

Proof: Using the definition of $\gg$ twice, we have $\epsilon.n \gg \pi = (\epsilon \gg \pi).n = \pi.n$.

Lemma 6.5 (prefix form) *If $\pi \in \mathit{Pos}$ is not ϵ, then there exist $\hat{n} \in \mathit{Nat}$, $\hat{\pi} \in \mathit{Pos}$ such that $\pi = (\hat{\pi} \gg \hat{n})$.*

Proof: By induction on the structure of π. The base case is trivial, so assume $\pi \equiv \pi'.n'$. If $\pi' \equiv \epsilon$, then ϵ and n' are the required $\hat{\pi}$ and $\hat{n}$. So assume $\pi' \not\equiv \epsilon$. Then by the induction hypothesis, there exist π'' and n'' such that $\pi' = \pi'' \gg n''$. Using the definition of $\gg$ and then Lemma 6.4, we have

$$(\pi'' \gg n'').n' = (\pi''.n' \gg n'') = (n' \gg \pi'') \gg n''$$

So $(n' \gg \pi'')$ and n'' are the required $\hat{\pi}$ and $\hat{n}$. $\Box$

Lemma 6.6 (commuting $\ll$ past ".") *For any $\pi \in$ Pos and $n_0, n_1 \in$ Nat, if $\pi \neq \epsilon$, then*

$$(\pi.n_0) \ll n_1 = (\pi \ll n_1).n_0$$

Proof: By Lemma 6.5, $\pi \equiv \pi' \gg n'$ for some π' and n'. Using the definition of $\gg$, we have

$$(\pi' \gg n').n_0 = \pi'.n_0 \gg n'$$

Then if $n' = n_1$, both $(\pi'.n_0 \gg n') \ll n_1$ and $((\pi' \gg n') \ll n_1).n_0$ are equal to $\pi'.n_0$ by definition of $\ll$. If $n' \neq n_1$, then n_1 cannot be a prefix of π, by properties of constructors. In that case, the terms to prove equal are both undefined. $\square$

Lemma 6.7 (associativity of $\gg$) *For any positions π_0, π_1, and π_2,*

$$(\pi_0 \gg (\pi_1 \gg \pi_2)) = ((\pi_0 \gg \pi_1) \gg \pi_2)$$

Proof: By induction on the structure of π_0.

Case $\pi_0 \equiv \epsilon$: Using the first clause of the definition of $\gg$ twice, we have $\epsilon \gg (\pi_1 \gg \pi_2) = (\pi_1 \gg \pi_2) = ((\epsilon \gg \pi_1) \gg \pi_2)$.

Case $\pi_0 \equiv \pi_0'.n$: Using the second clause of the definition of $\gg$, then the induction hypothesis, and then the second clause of the definition of $\gg$ again twice, we have

$$\begin{aligned} \pi_0'.n \gg (\pi_1 \gg \pi_2) &= (\pi_0' \gg (\pi_1 \gg \pi_2)).n \\ &= ((\pi_0' \gg \pi_1) \gg \pi_2).n \\ &= ((\pi_0' \gg \pi_1).n \gg \pi_2) \\ &= ((\pi_0'.n \gg \pi_1) \gg \pi_2) \quad \square \end{aligned}$$

Lemma 6.2 says: for $\Xi \in \{\lambda, \Pi\}$ and $i \in \{0, 1\}$,

$$\begin{array}{llll} i. & \Sigma \,|\, M_i \vdash a & \text{implies} & \Sigma \,|\, (M_0\ M_1) \vdash \ (a \gg i) \cup I_\Sigma(M_0\ M_1) \\ ii. & \Sigma \,|\, \sigma \vdash a & \text{implies} & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash \ (a \gg 1) \cup I_\Sigma(\Xi\, x : \sigma.\, M) \\ iii. & \Sigma, x : \sigma \,|\, M \vdash a & \text{implies} & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash \ (a \gg 2) \cup I_\Sigma(\Xi\, x : \sigma.\, M) \end{array}$$

Proof (Lemma 6.2): Let S be the sequent we must show is derivable. For each claim of the lemma, the proof proceeds by induction on the structure of the assumed derivation (call it d).

Proof (i): Let $i \in \{0, 1\}$ be arbitrary.

Case (ax): d proves $\Sigma \mid M_i \vdash I_\Sigma(M_i)$. S is an instance of (ax), since by definition of I_Σ, we have $I_\Sigma(M_0\ M_1) \supset (I_\Sigma(M_i) \gg i)$.

Case (lam): d ends in an inference of a sequent $\Sigma \mid M_i \vdash a \cup \{\pi : \Pi\, x : \sigma.\, \tau\}$ from sequents $\Sigma \mid M_i \vdash a$ and $\Sigma \mid \Pi\, x : \sigma.\, \tau \vdash b$, where

- a is $a' \cup \{\pi.2 : \tau\}$, for some a'
- b is $b' \cup \{\epsilon : \alpha\}$, for some b' and with $\alpha \in \{\mathit{type}, \mathit{kind}\}$
- x is $(M_i \ll \pi.0)$ and σ is $(M_i \ll \pi.1)$

By the induction hypothesis, there is a derivation d' of

$$\Sigma \mid (M_0\ M_1) \vdash (a \gg i) \cup I_\Sigma(M_0\ M_1).$$

Using Definition 6.5 ($\gg$ extended to functions) and then the definition of $\gg$, we see that this sequent equals

$$\Sigma \mid (M_0\ M_1) \vdash (a' \gg i) \cup \{(\pi \gg i).2 : \tau\} \cup I_\Sigma(M_0\ M_1)$$

From this sequent and $\Sigma \mid \Pi\, x : \sigma.\, \tau \vdash b$, (lam) may be used to conclude

$$\Sigma \mid (M_0\ M_1) \vdash (a \gg i) \cup \{(\pi \gg i) : \Pi\, x' : \sigma'.\, \tau\} \cup I_\Sigma(M_0\ M_1),$$

where x' is $((M_0\ M_1) \ll (\pi \gg i).0)$ and σ' is $((M_0\ M_1) \ll (\pi \gg i).1)$. But (lam) may be used here only if $x \equiv x'$ and $\sigma \equiv \sigma'$; otherwise, $\Sigma \mid \Pi\, x : \sigma.\, \tau \vdash b$ would not be a suitable premise for the new application of (lam). By the definition of $\gg$ on annotations, the derived sequent is equal to the sequent S whose derivability is to be shown.

To show $x' \equiv x$ and $\sigma' \equiv \sigma$, let $j \in \{0, 1\}$ and $\pi' \in \mathit{Pos}$ be arbitrary. Using Definition 6.5 ($\ll$ extended to functions), then the definition of $\gg$, then associativity of $\gg$ (Lemma 6.7), then Definition 6.5 again, then Notation 6.1 (expressions), and finally

Definition 6.5 again, we obtain

$$\begin{aligned}((M_0\ M_1) \ll (\pi \gg i).j)(\pi') &= (M_0\ M_1)(\pi' \gg (\pi \gg i).j)\\ &= (M_0\ M_1)(\pi' \gg (\pi.j \gg i))\\ &= (M_0\ M_1)((\pi' \gg \pi.j) \gg i)\\ &= ((M_0\ M_1) \ll i)(\pi' \gg \pi.j)\\ &= M_i(\pi' \gg \pi.j)\\ &= (M_i \ll \pi.j)(\pi')\end{aligned}$$

Case (app) and (pi): The proofs for these cases are similar to the one for (lam).

Proof (ii) and (iii): All cases are similar to the ones for (i). The cases for (ax) follow the definition of I_Σ, just as they did for (i). □

Lemma 6.3 says: for $\Xi \in \{\lambda, \Pi\}$,

$$\begin{array}{llll} i. & \Sigma \,|\, (M_0\ M_1) \vdash a & \text{implies} & \Sigma \,|\, M_i \vdash \quad (a \ll i),\ \text{for } i \in \{0,1\}\\ ii. & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash a & \text{implies} & \Sigma \,|\, \sigma \vdash \quad (a \ll 1)\\ iii. & \Sigma \,|\, \Xi\, x : \sigma.\, M \vdash a & \text{implies} & \Sigma,\, x : \sigma \,|\, M \vdash \quad (a \ll 2)\end{array}$$

Proof (Lemma 6.3): Let S and d be as in the proof of Lemma 6.2.

Proof (i): Let $i \in \{0, 1\}$ be arbitrary.

Case (ax): d proves $\Sigma \,|\, (M_0\ M_1) \vdash I_\Sigma(M_0\ M_1)$. S is an instance of (ax), since using the definitions of I_Σ and $\ll$, we have

$$(I_\Sigma(M_0\ M_1)) \ll i \quad = \quad (I_\Sigma(M_0) \gg 0) \cup (I_\Sigma(M_1) \gg 1)) \ll i \quad = \quad I_\Sigma(M_i)$$

Case (lam): d ends in an inference of a sequent

$$\Sigma \,|\, (M_0\ M_1) \vdash a \cup \{\pi : \Pi\, x : \sigma.\, \tau\}$$

from sequents $\Sigma \,|\, (M_0\ M_1) \vdash a$ and $\Sigma \,|\, \Pi\, x : \sigma.\, \tau \vdash b$, where

- a is $a' \cup \{\pi.2 : \tau\}$, for some a'
- b is $b' \cup \{\epsilon : \alpha\}$, for some b' and with $\alpha \in \{\textit{type}, \textit{kind}\}$

- x is $((M_0\ M_1) \ll \pi.0)$ and σ is $((M_0\ M_1) \ll \pi.1)$

By the induction hypothesis, there is a derivation d' of the sequent $\Sigma \mid M_i \vdash (a \ll i)$. If $\pi = \epsilon$, then this is already equal to the desired sequent S. So suppose $\pi \neq \epsilon$. Now by Definition 6.5 ($\ll$ extended to functions), $\Sigma \mid M_i \vdash (a \ll i)$ is equal to

$$\Sigma \mid M_i \vdash (a' \ll i) \cup \{(\pi.2 \ll i) : \tau\}$$

By Lemma 6.6 (which applies since $\pi \neq \epsilon$), $(\pi.2 \ll i) = (\pi \ll i).2$. If $(\pi \ll i) \uparrow$, $(a' \ll i)$ is already equal to $(a \ll i)$. So suppose $(\pi \ll i) \downarrow$. Then (lam) may be used with this sequent and $\Sigma \mid \Pi\, x : \sigma.\, \tau \vdash b$ to conclude

$$\Sigma \mid M_i \vdash (a \ll i) \cup \{(\pi \ll i) : \Pi\, x' : \sigma'.\, \tau\},$$

where x' is $(M_i \ll (\pi \ll i).0)$ and σ' is $(M_i \ll (\pi \ll i).1)$. This application is legal, and this last sequent is equal to the sequent S whose derivability is to be shown, only if $x' \equiv x$ and $\sigma' \equiv \sigma$. This is proved as follows. Let $j \in \{0, 1\}$ be arbitrary. Since $(\pi \ll i) \downarrow$, $\pi = (\pi' \gg i)$, for some π'. So it is necessary to prove

$$((M_0\ M_1) \ll (\pi' \gg i).j) = M_i \ll ((\pi' \gg i) \ll i).j,$$

which simplifies to $((M_0\ M_1) \ll (\pi' \gg i).j) = M_i \ll \pi'.j$, which was already proven in the corresponding case for Lemma 6.2.

Cases (app) and (pi): The proofs for these cases are similar to the one for (lam).

Proof (ii) and (iii): All cases are similar to the ones for (i). □

6.3 Context-dependent classifier caching

This section presents the formalization of the approach to caching and reusing computed classifiers described at the start of the chapter, and proves it sound. Some preliminary definitions are needed.

Definition 6.7 (*FS*(e) : free symbols of e) *The function FS computing the set of symbols* *occurring freely in* $e \in Exp$ *is defined by:*

$$\begin{array}{rcl} \mathit{FS}(\mathit{type}) & = & \mathit{FS}(\mathit{kind}) = \emptyset \\ \mathit{FS}(M\ N) & = & \mathit{FS}(M) \cup \mathit{FS}(N) \\ \text{for } \Xi \in \{\lambda, \Pi\}, \mathit{FS}(\Xi\, x : \sigma.\, M) & = & \mathit{FS}(\sigma) \cup (\mathit{FS}(M) \setminus \{x\}) \\ \text{for } v \in \mathit{Sym}, \mathit{FS}(v) & = & \{v\} \end{array}$$

Definition 6.8 ($\mathit{Ctx}_e(\pi)$: context at π in e) *Let* $e \in Exp$, $\pi \in \mathit{Def}(e)$, *and* $n \in Nat$. $\mathit{Ctx}_e(\pi)$ *is the list of bindings at proper prefixes of* π *in* e*:*

$$\begin{array}{rcl} \mathit{Ctx}_e(\epsilon) & = & \cdot \\ \mathit{Ctx}_e(\pi.n) & = & \begin{cases} \mathit{Ctx}_e(\pi), (e \ll \pi.0) : (e \ll \pi.1) & \text{if } e(\pi) \in \{\lambda, \Pi\} \text{ and } n = 2 \\ \mathit{Ctx}_e(\pi) & \text{otherwise} \end{cases} \end{array}$$

Example 2 *Let* e *be* $\lambda\, x : \sigma.\, \lambda\, y : \sigma.\, \lambda\, x : \tau.\, x$. *Then*

$$\mathit{Ctx}_e(2) = x : \sigma \quad \mathit{Ctx}_e(2.2.2) = x : \sigma,\ y : \sigma,\ x : \tau$$

Definition 6.9 ($\mathit{Dep}_e(\pi)$: dependencies at π in e) *Let* $e \in Exp$ *and* $\pi \in \mathit{Def}(e)$. *Then*

$$\mathit{Dep}_e(\pi) = \{x : \mathit{lookup}(x, \mathit{Ctx}_e(\pi)) \mid x \in \mathit{FS}(e \ll \pi)\}.$$

Intuitively, the intention is that $\mathit{Dep}_e(\pi)$ be the set of all those bindings in $\mathit{Ctx}_e(\pi)$ that the classification of $(e \ll \pi)$ depends on.

Example 3 *Let* e *be as in Example 2. Then* $\mathit{Dep}_e(2.2.2) = \{x : \tau\}$.

Now consider the extension of the annotation calculus of Figure 6.2 by the following new rule:

$$\text{(copy)} \quad \frac{\Sigma \,|\, e \vdash a}{\Sigma \,|\, e \vdash a \cup ((a \ll \pi) \gg \pi')} \quad \begin{array}{l} (e \ll \pi) = (e \ll \pi') \\ \mathit{Dep}_e(\pi) = \mathit{Dep}_e(\pi') \end{array}$$

Suppose subexpression $(e \ll \pi)$ of e occurs at position $\pi' \neq \pi$ (as well as at position π). Then informally, the (copy) rule says that if the annotation for the subexpression at the first position has been computed, then it can be simply copied in a single step to annotate that subexpression at the second position (π'), as long as the dependencies of the subexpression are the same at the two positions.

Example 4 (derivation using (copy)) *Let Σ be a signature*

$$\tau : \mathit{type},\ f : \tau \to (\tau \to \tau),\ g : \tau \to \tau,\ \ldots.$$

Let e be $\lambda\, x : \tau.\, (f\ (g\ (g\ x)))\ (g\ (g\ x))$. The following derivation uses (copy) to avoid recomputing the annotation for the second occurrence of $(g\ (g\ x))$. Note that the position of the first occurrence of that expression is $2.0.1$ and the position of the second occurrence is 2.1. We abbreviate the derivation by showing just how annotations are extended. The rules used are, from top to bottom, (ax), (app), (app), and (copy). In the instance of (copy), $\pi = 2.0.1$ and $\pi' = 2.1$. Note that (copy) extends the annotation at several positions, because (copy) copies the entire annotation of one subexpression to another. So a single application of (copy) annotates at the second occurrence of $(g\ (g\ x))$ all those subterms which are annotated at the first occurrence.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\Sigma \,|\, e \vdash \{\ldots,\ 2.0.1.0 : \tau \to \tau,\ 2.0.1.1.0 : \tau \to \tau,\ 2.0.1.1.1 : \tau\}}}{2.0.1.1 : \tau}}{2.0.1 : \tau}}{2.1.1 : \tau,\ 2.1 : \tau}$$

Theorem 6.2 (conservativity of (copy)) *A sequent is derivable in the calculus with the (copy) rule iff it is derivable in the original calculus without (copy).*

Proof: The "if" direction of the proof is obvious. For the "only if" direction, the proof is by induction on the assumed derivation with (copy). All cases follow easily using the induction hypothesis, except the case for (copy). In that case, by the induction hypothesis, we have a derivation without (copy) of $\Sigma \,|\, e \vdash a$. We must show that $\Sigma \,|\, e \vdash a \cup ((a \ll \pi) \gg \pi')$ is derivable without (copy), where $(e \ll \pi) = (e \ll \pi')$ and $\mathit{Dep}_e(\pi) = \mathit{Dep}_e(\pi')$.

Let $C = \mathit{Ctx}_e(\pi)$ and $C' = \mathit{Ctx}_e(\pi')$. Observe first that

$$\Sigma \,|\, e \vdash a \ \text{ implies } \ \Sigma, C \,|\, (e \ll \pi) \vdash (a \ll \pi).$$

This follows readily by induction on π using Lemma 6.3 (shifting). Then we can show that $\Sigma, C' \mid (e \ll \pi') \vdash (a \ll \pi)$ is derivable. This is proved by showing that $I_{\Sigma,C}(e \ll \pi) = I_{\Sigma,C'}(e \ll \pi')$, which is done readily by induction on the structure of $(e \ll \pi)$: all cases go through easily using the induction hypothesis, except for the case when $(e \ll \pi)$ is a symbol, where the fact that $\mathit{Dep}_e(\pi) = \mathit{Dep}_e(\pi')$ is used. Finally, it can then be proved by induction on π' using Lemma 6.2 (shifting) that

$$\Sigma, C' \mid (e \ll \pi') \vdash (a \ll \pi) \text{ implies } \Sigma \mid e \vdash ((a \ll \pi) \gg \pi') \cup I_\Sigma(e)$$

The result now follows using Lemma 6.1 (monotonicity). □

6.3.1 Implementation

The (copy) rule forms the theoretical basis for context-dependent caching. One way to implement (copy) would be to maintain a hash table H mapping a pair of the form $\langle (e \ll \pi), \mathit{Dep}_e(\pi) \rangle$ to the classifier computed for $(e \ll \pi)$ at a position with dependencies $\mathit{Dep}_e(\pi)$. This would result in the maximum reusing of computed classifiers allowed by (copy). The table H, however, could become very large, with no guarantee that cached results would be reused frequently. The flea proof checker implements a more memory-efficient approximation to this scheme; the approximation seems to work well in practice, although there are cases where it achieves less reuse than the full caching scheme. We approximate $\mathit{Dep}_e(\pi)$ by the position of e's deepest relevant binding. More precisely, we approximate $\mathit{Dep}_e(\pi)$ by the longest prefix ψ of π in e such that the binding $(e \ll \psi.0) : (e \ll \psi.1)$ is in $\mathit{Dep}_e(\pi)$. This position ψ is called the *context id* of the occurrence of $(e \ll \pi)$ at π. Now H is taken to be a hash table from subexpressions $(e \ll \pi)$ to pairs $\langle \tau, \mathit{id} \rangle$, where τ is a classifier and *id* is the context identifier of the occurrence of $(e \ll \pi)$ for which τ was computed. Before trying to compute the classifier for a subexpression s, H is consulted to see whether or not there is a cached classifier τ for s which has the same context identifier as this occurrence of s. If so, τ is used for this occurrence. If not, the classifier τ' for s at this occurrence is computed, and H is modified to map s to $\langle \tau', \mathit{id} \rangle$, where *id* is the context id for s at this occurrence. See Section 6.6 for empirical results.

6.3.2 Proof compression

We briefly sketch an approach based on context-dependent caching to proof compression. An expression e is compressed as follows. With each subexpression $(e \ll \pi)$ at position π, we cache in a context-dependent way an abbreviation

$$a := \lambda\, x_1 : \tau_1. \ldots \lambda\, x_n : \tau_n. (e \ll \pi)$$

where $\mathit{Dep}_e(\pi) = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ and a is some new symbol. We then transform the subexpression $(e \ll \pi)$ by replacing it with $(\ldots(a\ x_1) \ldots\ x_n)$. If the type checker permits, the abbreviation can be inserted at the position in the transformed expression corresponding to the context id of this subexpression at this position π. More work is required to pull the abbreviation all the way to the front of the transformed expression. This is because the types $\tau_1, \ldots, \tau_n$ may contain free variables besides the ones $x_1, \ldots, x_n$ accounted for in the body of the abbreviation. To pull the abbreviation upward in the expression across a λ- or Π-binding of a variable v, if the body of the abbreviation as it currently stands contains v free, then that body will have to be augmented by a λ-abstraction of v. Applications of the new constant a will then have to take in v as another argument.

6.4 Variable-sensitive safe substitution

Profiling runs of flea checking proofs produced by CVC typically show that a large fraction of the time (around 30% on the dlx-pc example of Section 6.6) is spent performing safe substitutions. In this section, we describe some improvements to safe substitution. We define a substitution to be a function $\sigma : \mathit{Sym} \to \mathit{Exp}$ with a finite domain of definition $\{v_1, \ldots, v_n\}$. It may be denoted $\{v_1 := \sigma(v_1), \ldots, v_n := \sigma(v_n)\}$, possibly without the curly brackets. Let $\Xi \in \{\lambda, \Pi\}$ be arbitrary. Then the safe application $e[\sigma]$ of such a substitution σ to $e \in \mathit{Exp}$ is defined by induction on the structure of e. In the second case for λ-abstractions, $\hat{x}$ is a symbol not occurring in M or the range of σ.

$$\begin{aligned} \alpha[\sigma] &= \alpha, \text{ for } \alpha \in \{\mathit{type}, \mathit{kind}\} \\ (M\ N)[\sigma] &= (M[\sigma]\ N[\sigma]) \end{aligned}$$

$$(\Xi\, x : A.\, M)[\sigma] \;=\; \begin{cases} (\Xi\, x : A[\sigma].\, M[\sigma']) & \text{if } \sigma = (\sigma' \cup \{x := t\}) \\ & \text{ for some } t \\ \Xi\, \hat{x} : A[\sigma].\, M[\sigma \cup \{x := \hat{x}\}] & \text{if } x \in \mathit{FS}(\sigma(v)) \text{ for some } v, \\ \Xi\, x : A[\sigma].\, M[\sigma] & \text{otherwise} \end{cases}$$

$$\text{for } v \in \mathit{Sym},\, v[\sigma] \;=\; \begin{cases} \sigma(v) & \text{if } \sigma(v) \downarrow \\ v & \text{otherwise} \end{cases}$$

Safe substitution requires sets of free symbols of expressions to be computed. If the symbols bound by signature Σ are disjoint from the symbols bound by all Π- and λ-abstractions, then instead of $\mathit{FS}(e)$ it can easily be shown that we can use the set of free variables of e (relative to Σ), defined by

$$\mathit{FV}_\Sigma(e) = \{v \in \mathit{FS}(e) \mid \Sigma \text{ does not bind } v\}.$$

Furthermore, $\mathit{FV}_\Sigma(e)$ can be cached with e, so it need not be recomputed. Also, it is to be expected that many expressions will have the same set of free variables. A set S of the sets of free variables is maintained, and memory is allocated for a new set of free variables only if a set storing exactly those variables is not already in S. Finally, if $\mathit{FV}_\Sigma(e) \cap \mathit{Def}(\sigma) = \emptyset$, then an easy inductive proof shows that $e[\sigma] = e$. So there is no need to compute $e[\sigma]$ if this intersection is empty.

6.4.1 de Bruijn indices

The optimizations of safe substitution are described for the version of LF that uses named variables; this is what is implemented in flea. An alternative is to use de Bruijn indices for variables (see, e.g., [Kam]). The optimization of returning e for $e[\sigma]$ without actually applying the substitution in the case where $\mathit{FV}_\Sigma(e) \cap \mathit{Def}(\sigma) = \emptyset$ is still relevant with de Bruijn indices. The definition of $\mathit{FS}(\Xi\, x : \sigma.\, M)$ for $\Xi \in \{\lambda, \Pi\}$ is changed to be the following:

$$\mathit{FS}(\Xi\, x : \sigma.\, M) \;=\; \mathit{FS}(\sigma) \cup \{v - 1 | v \in \mathit{FS}(M)\}$$

The optimization is performed by checking the intersection of *FS*(e) and *Def*(σ) for emptiness. It would then make sense to share sets of free variables, as proposed in the case of named variables. We might expect many fewer different sets of de Bruijn indices, and so the optimization of using a set of sets of free variables could be even more effective. Furthermore, sets of de Bruijn indices can be compactly represented as bitvectors.

6.4.2 Explicit substitutions

A further improvement, which was also not implemented in flea, would be to use explicit substitutions and cache the result of computing $e[\sigma]$. This could be done by maintaining a hash table mapping expressions $e[\sigma]$ to the results of carrying out the substitutions. Using such a caching scheme could help improve performance, although it might be necessary to use a cache of fixed maximum size and implement some cache replacement policy to keep from using too much memory.

6.5 Classifier computation for left-nested applications

A left-nested application is an application like $(((c\ a_1)\ a_2)\ a_3)$. Suppose the classifier for a classifiable left-nested application $(\ldots(c\ a_1)\ldots\ a_n)$ is to be computed. Assuming that classifiers $\tau_1', \ldots, \tau_n'$ for $a_1, \ldots, a_n$ and $\Pi\, x_1\ :\ \tau_1.\ \ldots \Pi\, x_n\ :\ \tau_n.\,\tau$ (call this *optype*) for c have already been computed, the derivation in the annotation calculus (with positions written in bold for readability) would be

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\mathbf{0^{n-1}.0} : \Pi\, x_1 : \tau_1.\ \ldots \Pi\, x_n : \tau_n.\,\tau, \quad \mathbf{0^{n-1}.1} : \tau_1'}}{\mathbf{0^{n-2}.0} : \Pi\, x_2 : \tau_2[x_1 := a_1].\ \ldots \Pi\, x_n : \tau_n[x_1 := a_1].\,\tau[x_1 := a_1], \quad \mathbf{0^{n-2}.1} : \tau_2}}{\vdots}}{\mathbf{0} : \Pi\, x_n : \tau_n[x_1 := a_1]\ldots[x_{n-1} := a_{n-1}].\,\tau[x_1 := a_1]\ldots[x_{n-1} := a_{n-1}], \quad \mathbf{1} : \tau_n}}{\epsilon : \tau[x_1 := a_1]\ldots[x_n := a_n]}$$

where the following definition has been used:

Definition 6.10 (iterated ".") *For* $n, m \in$ *Nat* , n^m *is defined inductively by* $n^0 = \epsilon$ *and* $n^{m+1} = (n^m).n$

Several things may be observed about how substitutions are carried out in this derivation. First, $\Omega(n^2)$ time is spent doing substitutions, since in the i'th step of the derivation, a substitution is performed on $(\mathit{optype} \ll 2^i)[x_1 := a_1] \ldots [x_{i-1} := a_{i-1}]$. This last expression is at least as big as $(\mathit{optype} \ll 2^i)$, which is of size at least $n - i$. There are n steps in the derivation, so the total time is at least quadratic. Second, consider what happens in, for example, step 3 of such a derivation. If τ_3 has an occurrence of x_1 in it, then computing $\tau_3[x_1 := a_1][x_2 := a_2]$ will require computing $a_1[x_2 := a_2]$, even though a_1 cannot possibly contain x_2 free. Variable-sensitive substitution would keep the second observation from leading to inefficiency, but not the first; the following solution prevents both problems. The annotation calculus's (app) rule is replaced by the rule (app') below. For notational simplicity, we give just the instance of the rule that extends the annotation at position ϵ, instead of at an arbitrary position. For $i \in \{1, \ldots, n\}$, let a_i abbreviate $(e \ll 0^{n-i}.1)$, which is the i'th innermost argument in the nested application.

$$\text{(app')} \quad \frac{\mathbf{0^{n-1}.0} : \Pi\, x_1 : \tau_1. \ldots \Pi\, x_n : \tau_n. \tau, \quad \mathbf{0^{n-1}.1} : \tau_1', \ \ldots, \ \mathbf{1} : \tau_n'}{\epsilon : \tau[x_1 := a_1, \ldots, x_n := a_n]} \quad \phi$$

The side condition ϕ on the rule is the conjunction of the following:

- $e(0^i) = @$, for all $i \in \{0, \ldots, n-1\}$
- $\tau_1 \cong \tau_1'$
- for $i \in \{2, \ldots, n-1\}$, $\tau_i[x_1 := a_1, \ldots, x_{i-1} := a_{i-1}] \cong \tau_i'$

In the case where for some $1 \leq i < j \leq n$, $x_i \equiv x_j$, we stipulate that the binding of x_i occurring further to the left in the substitution is dropped from the substitution. If the types $\tau_1, \ldots, \tau_n$ are each of constant size, then this rule (whose proof of correctness we omit to save space) requires only $O(n)$ time to be spent performing substitutions. This is because the side condition applies a substitution (of size $O(i)$) n times to something of constant size.

This optimization applies larger substitutions to smaller number of subterms. With larger substitutions, the following two tasks will be more expensive: performing the intersection check described in Section 6.4 and when substituting into an abstraction, checking whether or not a term in the range of the substitution has the variable bound in the abstraction free. The empirical results in the next section show that in some cases it may be more efficient to apply larger substitutions to fewer terms and in other cases applying smaller substitutions to more terms may be more efficient. We do not currently have a characterization of when one approach is better than the other.

6.6 Empirical results

This section gives some empirical results on proofs produced by CVC for formulas generated from verification problems. The proofs use some extensions to LF for dealing with lists more directly. In order to compare with two other systems implementing LF type-checking, a large piece of one example was hand-translated to remove the extensions, and common subexpressions were pulled out using abbreviations (see Section 6.3.2); dlx-pc.pure is the result. This example could then be run through Twelf [PS99] and, with a minor translation, LEGO [LP92].[1]

Figure 6.3 gives the results. Entries in the table show time and peak memory usage on a 850MHz Pentium III with 256M of main memory. "all optimizations" is the flea proof checker with all the optimizations described above. "-distinct vars" is "all" except without the optimization allowed by keeping bound variables distinct from constants declared in the signature (Section 6.4). "-intersection check" is "all" without the check on the intersection of domain of definition of substitution and set of free variables of expression (Section 6.4). "-fv sets" is "all" except that memory for a set of free variables is allocated without consulting a set of sets of free variables (Section 6.4). "-left nest" is "all" except without the modification to allow efficient computation of the classifier of left-nested applications (Section 6.5). Example size is the size in ASCII text with maximal sharing of common subexpressions. No results are given in the table for checking without context-dependent caching, because as mentioned earlier, checking is not feasible without this optimization. As one example of this, a small subproof of dlx-pc of size 90K takes 1.3s to check with context-dependent caching and over 1 minute to check without. This is because as a tree, the subproof is 3.5M long.

Without "intersection check", checking time varies rather widely, possibly because the intersection check ameliorates some of the inefficiency of performing substitution over

[1]The example in pure LF is available on the web in Twelf and LEGO formats at http://verify.stanford.edu/~stump/dlx-pc.pure/. Also, the flea proof checker ships with CVC, which is freely available at http://verify.stanford.edu/CVC/.

	dlx-pc.pure	dlx-pc	satyaki5	pp-invariant
all optimizations	4.4s, 31M	24.7s, 49M	21.6 s, 36M	68s, 68M
-distinct vars	8.9s, 42M	46.7s, 65M	47.6s, 39M	97s, 95M
-intersection check	4.0s, 31M	>1000s, >256M	720s, 167M	127s, 72M
-fv sets	4.1s, 32M	22.3s, 60M	19.9s, 42M	84s, 90M
-left nest	6.4s, 60M	21.3s, 72M	24.5s, 48M	53.5s, 94M
Twelf	26.8s, 84M	[examples not in pure LF format]		
LEGO	930s, 51M	[examples not in pure LF format]		
example size	2.4M	2.2M	0.9M	4.6M

Figure 6.3: Empirical results comparing optimizations

a DAG without caching. The results of substituting cannot be easily cached, because the substitution being applied changes; as explained in Section 6.4 above, using explicit substitutions would help with this problem.

6.7 Performance of proof production and proof checking

In Figure 6.4, the times required to produce proofs from CVC and the times required to check them with flea are compared. The same machine was used for these experiments as for those of the previous section. Checking proofs takes substantially longer than producing them, even with the optimizations developed in this chapter. This was originally quite surprising, but it should be considered that CVC is not searching for a proof (among, say, a space of possible proofs), but constructing one according to rules that exactly determine the next inference to be applied at every step. While CVC can construct each step of the proof simply by building the appropriate LF term, flea must check that that term is well-typed. This puts a significant computational burden on flea. The difference is not between finding a proof and checking it, but between deterministically constructing a proof and checking it. Viewed this way, it is not so surprising that producing a proof takes much less time than checking it.

example	time to produce	time to check
dlx-pc	4.5s	24.7s
satyaki5	1.6s	21.6s
pp-invariant	20s	68s
arith_nikolaj	0.3s	1.8s
tiwari4	0.1s	1.0s
array_equations2hard	0.2s	1.1s

Figure 6.4: Comparison of time to produce proofs and time to check them

6.8 Conclusion

Several optimizations for LF type-checking have been presented. To formalize context-dependent caching, a path calculus for LF has been developed and proven equivalent to a standard calculus. Optimizations related to safe substitution and classifier computation for left-nested applications have also been presented. Further empirical work is needed to see which optimizations are most important for particular classes of represented proofs. The results obtained here should generalize to Pure Type Systems (PTSs), although it is not obvious how to extend caching to PTSs that are not singly sorted, since unicity of classifiers can fail (see Lemma 5.2.21 of [Bar92]). For similar reasons, it is not clear to what extent context-dependent caching can be used in type-checking implicit LF [NL98] or systems of the Rho Cube [CKL01b].

Chapter 7

Implementation of CVC and flea

This chapter discusses selected aspects of the implementation of CVC and flea, which is publicly available at http://verify.stanford.edu/CVC/. The codebase for CVC and flea is around 150Kloc of C++. The approximately 16Kloc of code for the SAT solver and associated glue code are written by others (namely, the Chaff developers and Clark Barrett). All the other code is by the author. We begin with a discussion of the component-oriented programming style used in CVC and flea.

7.1 Component-oriented programming in C++

The basic idea of object-oriented programming in C++ is roughly that classes that should be defined which correspond to the natural kinds of entities being considered. For example, in writing a program to help manage an inventory of car parts, one might have classes like Tire or Muffler. These classes could then be subclassed by classes like AllWeatherTire or OffRoadTire, in an attempt to reflect a perceived natural hierarchy of entities in the problem domain as a class hierarchy in the C++ code. Modeling a problem domain as an object-oriented system can be done by "finding the nouns" in the problem domain and making them into classes. Entities in the problem domain get modeled as data objects, and any functions that operate on those data objects are associated with them as methods of their classes. This is the classical object-oriented approach to design.

If object-oriented design emphasizes modeling the nouns in a problem domain, then component-oriented design seeks to model the verbs. In a component-oriented approach to design, the goal is to model the operations that need to be performed to solve a problem or provide a service as components. There are two main advantages to this approach. First, functions (or groups of functions) that need to maintain some data across calls to do their job can have that data associated with them as member data of their classes. They do not need to use global variables for this purpose. Note the duality with the case of data objects, where we think primarily of the member data of the class and secondarily of associating functions operating on that data as methods of the class. In component-oriented programming, we think primarily of the methods, and secondarily of the member data.

The second advantage to the component-oriented approach is that by grouping functions into classes, interface classes for components can be defined. An interface class just lists methods that any subclass implementing the interface must implement. Other components may then be written purely in terms of interface classes. Any concrete implementation (i.e., instance of a concrete subclass) of the interface class can then be used by such components. This component-oriented style of design results in systems consisting of loosely coupled components which interact with each other through interfaces. Such loosely coupled designs are more flexible, reusable, and maintainable than classical object-oriented designs. We should note, however, that component-oriented programming just adds to the concepts of classical object-oriented programming the concept of a component. A good design, particularly for a relatively large system like CVC, will probably contain both data objects and components operating on those data objects.

For an example, consider an application for viewing and converting images. A classical object-oriented design might have C++ classes like JPEG_Image and TIFF_Image. These classes might have constructors taking in raw displayable data and creating an image of the appropriate format from it. They might also have ConvertToRaw() methods to convert to raw data. Images could be converted from one format to another using raw data as an intermediary format. It can be much more efficient, however, to convert directly from one image format to another without going through raw data format. For example, JPEGs can sit directly in TIFF files. It would be needlessly inefficient to render a JPEG to raw data and then put that data in TIFF format. We could improve performance in this design by adding a method ConvertToTIFF() to JPEG_Image. But notice what happens now. Every time support for a new image format F is added, we will have to add ConvertToF() methods to the classes for the various other image formats which can be efficiently converted directly to format F. So the definitions of the classes for the image formats will get larger and larger. Furthermore, it will become inconvenient to ship versions of the program with some image formats disabled (which might be desirable if a third-party library is used to support one of the formats, say).

A component-oriented design for this problem begins just the same way as the object oriented design. We have data objects for images in the different files formats. But then rather than add conversion methods to the classes for those file formats, we create a new class Render_X_To_Y whenever we wish to support direct conversion from file format X to Y. Just as in the object-oriented design, of course, we will get $O(n^2)$ different conversion functions if we support direct conversion between every pair of file formats. But notice that in this design, each such conversion routine goes into its own brand new class. The old classes for the different image formats and other conversions do not need to be modified at all. This component-oriented design is superior to the classical object-oriented one for several reasons. At the cost of the small code overhead of putting the different conversion routines in their own classes, we get a design where new functionality is added by adding new classes rather than by modifying old ones. The ability to evolve a codebase by addition rather than by modification is highly desirable, since modifications to existing classes can easily introduce errors into code that might have been error-free before. Furthermore, it is much more convenient to omit certain classes from versions of a program than it is to omit certain methods from a single class. Hence, the existing functionality supported by this component-oriented design can be more flexibly altered or configured than in the classic object-oriented design.

7.2 About the codebase

As mentioned at the start of the chapter, CVC's codebase is around 150Kloc of C++ code. The component-oriented methodology led in this case to a large number of source files (1378) with a very small average length (109 lines). The codebase adheres in most cases to the convention of having a separate .h and .cpp file for each class, so the classes are on average quite small. This suggests that the code for each class by itself should be easily understandable, as long as the dependencies between classes are not too many or too complex. In terms of the component-oriented approach to design just discussed, there is really just one kind of data object, namely symbolic expressions. Almost all other objects are components implementing various operations on expresssions, like rewrites or Tarjan's union-find operations [Tar75].

7.3 Parsing and pre-processing in CVC

This section describes part of the architecture for an important subsystem of CVC, the parser and pre-processer. This subsystem is relatively easy to explain, while still demonstrating the component-oriented design ideas followed throughout the system. References to places in the codebase are included in parentheses for the benefit of future developers of CVC. The code need not be consulted to follow the discussion. (By default, references to the codebase are to locations in cvc/src.)

One general architectural note is required before considering the particular subsystem. Different large subsystems of CVC export components for use by other subsystems. Components are exported by exporting a single Component_Registry (nucleus/ component_registry.h in Core_Utilities/src), which maps numeric IDs to Components (nucleus/ component.h in Core_Utilities/src). This requires exported components to be instances of subclasses of Component. A downcast is used to convert the Component returned for a particular numeric ID to the correct type. The numeric IDs for which a subsystem provides Components in this way are all named. Reference will be made in square brackets to the names of those IDs. This is also for the benefit of future developers of CVC. Such names are qualified by the namespace in which the exporting subsytem is defined (see cvc/src/ INDEX_OF_PACKAGES.txt).

CVC operates by reading and processing commands one at a time from standard input. Commands include things like declarations of types and constants; ASSERT commands, which request CVC to add a formula to the current logical context; and QUERY commands, which request CVC to check the validity of a formula in the current logical context. The parser (lang/PL.y) parses a single command by building an Expr representing it. Exprs are data structures representing abstract syntax trees as directed acyclic graphs (DAGs), with maximal sharing of common subexpressions. The Expr_Package part of the codebase implements this data structure, together with features like automatic storage reclamation for Exprs and the ability to reserve and then manipulate attributes on Exprs (see src/ expr_manager.h in Expr_Package).

Once a command has been parsed into an Expr, it is passed along to an instance of a subclass of Command_Processor [Lang::CMD_PROC] (see lang/ lang.cpp). Command_Processor is an interface class with a single method, Evaluate(), that takes in a single Expr as input. The particular Command_Processor to which the command is passed is a Transforming_Command_Processor (cmd_interpreter/ transforming_command_processor.h in Expr_Utilities/src). A Transforming_Command_Processor transforms a command using a Transformer. Transformer (nucleus/transformer.h in Core_Utilities/src) is an interface class with a single Transform() method, that takes in one piece of Data (a template parameter) as input and returns another as output. Transforming_Command_Processor's constructor takes in the Transformer (of Exprs) to use, as well as another Command_Processor, to which the result of transforming the command to process is then given.

7.3.1 Transforming the command

The particular Transformer used to transform a parsed command performs two steps of preprocessing, each of which is implemented as its own Transformer. The first step [Lang:: PRES_TO_INT] is to translate the command from the presentation language of CVC to the internal language. The translation removes some syntactic sugar in the presentation language. For example, the presentation language has a single "WITH" construct that can functionally update arrays, elements of user-declared abstract data types, records, and tuples; the internal language has different operators for updating each of those different data structures. The second step of preprocessing [Lang::TRANS_INT] is to perform various transformations on the resulting command in the internal language. An example is the removal of LET expressions, which introduce local abbreviations for expressions. LET expressions are eliminated by replacing abbreviations with the Exprs they abbreviate. This does not result in a blow-up in the size of the Expr, because the Expr_Package shares all common subexpressions. The two steps of preprocessing are combined using a Composing_Transformer (transforms/ composing_transformer.h in Core_Utilities/src), which just composes Transformers. Various hooks are provided that allow other subsystems to register Transformers to be called in one or the other step of preprocessing. Two commonly used hooks [Lang:: REG_TRANS_INT and Lang:: REG_PRES_TO_INT] allow Transformers to be registered for particular built-in operators (like PLUS or RECORDTYPE). Those Transformers will be called to transform Exprs built using those operators, in either the first or second step of preprocessing (depending on whether they were registered with REG_PRES_TO_INT or REG_TRANS_INT).

7.3.2 Processing the transformed command

Once a command has been pre-processed using the Transformers just described, it is given by the Transforming_Command_Processor to another Command_Processor, which will actually process the transformed command. Commands are processed by dispatching them to yet other Command_Processors, which are registered [with Lang:: REG_CMD_PROC] for the operator of the command. The subsystem for CVC's types (types/types.h) registers several such Command_Processors to handle the commands which declare types and constants. Command_Processors are also registered to handle ASSERT and QUERY commands.

7.3.3 Defense of registration

The design pattern of registering different independent components to handle different kinds of data is used heavily in the components just described for transforming and processing commands. To some, this approach may seem overly complicated, but it is actually very simple and clean to use. For example, to transform scalartypes like " a, b, c" to the appropriate flat datatype with 0-ary constructors a, b, and c, the decision procedure for abstract datatypes just uses the following code to register a suitable transformation:

```
Transform_Scalartypes *trans_scalartypes =
  (new Transform_Scalartypes)->Init(em);

reg_pres_to_int->Register(CVC::SCALARTYPE,
                          trans_scalartypes,
                          true);
```

Transform_Scalartypes is a tiny class defined in 53 lines of code. The third argument to Register() tells reg_pres_to_int to take responsibility for trans_scalartype's memory.

A seemingly simpler approach to adding new transformations might have a single class called Transform_Pres_To_Int, which would contain the code for all the different transformations required to convert from the presentation language to the internal language. But this class would have to be modified every time a new transformation was to be added. In addition to leading to a longer and longer source file for the class, this approach would lead to a class that would be subject to frequent modifications. So on this approach, classes would be less stable, in the sense that they would change more frequently than they would under the approach using registration. In that approach, the codebase changes mostly by addition of new classes, rather than modification of old ones. As remarked previously, it is generally preferable to build the codebase by addition rather than modification to avoid introducing new bugs into code which may have been bug-free before.

Chapter 8

Conclusion

This thesis has described solutions to crucial technical problems that had to be solved to implement the proof-producing cooperating validity checker CVC. The result is a fully automatic reasoning engine that can be used as a high-performance, flexible, and extensible verification tool. The ability of CVC to produce proofs that can be reasonably efficiently checked by the flea proof checker can greatly increase confidence in the output of CVC by providing a much simpler independent check on that output.

8.1 Future work

Much future work remains to be done for both CVC and flea. For CVC, it would be very useful to have a high-level language for describing subsidiary decision procedures. The goal would be to have a language where such procedures could be described in a notation very close to that used for proving them correct, but then could be compiled down to C++ code automatically. This would make it easier to add new decision procedures to CVC, and help prevent errors that could arise when developers have to implement abstract descriptions of algorithms. A suitable high-level language might be a basic rewriting language or perhaps something like the Rho Calculus [CKL01a].

On the flea side, work could proceed in several directions. First, further investigation into meta-languages for describing proofs is warranted. The Edinburgh LF is a well understood and simple meta-language, but its limitations required extensions, including a weak form of predicate subtyping, to be easily useable for representing proofs from CVC.

Those extensions were shown to render the system undecidable, although this was shown not to be a problem in practice. One idea would be to look for decidable fragments of LF extended with some form of predicate subtyping. This would lead to a more powerful, less easily checked meta-language. In fact, if the predicates allowed in the predicate subtypes were from the theories decided by CVC, checking a proof might require calls to CVC! But since CVC produces proofs, those calls would generate simpler proofs that could still by checked by the current version of flea. Having a proof language like this would allow proofs to contain gaps that would be filled in automatically by the proof-checker. This could be attractive for applications like proof-carrying code, where it is important to keep proofs as small as possible.

Instead of making the meta-language for proofs as powerful as possible, one could try to make it weaker. The goal here would be to have the smallest possible proof checker. The author was able to create a prototype LF checker in 282 lines of C code, without a parser. A parser and a few other features were added to this checker by researchers at Princeton, resulting in an LF checker that is 803 lines of C code long [AMSV02]. It should be possible to get this down significantly by simplifying the proof language. For example, it seems as though it should be possible to check resolution proofs with a very tiny proof checker. LF proofs could still be supported by such a checker, in the following way. It should be possible to write down first order axioms for LF using techniques like explicit substitutions. One could then have an LF checker that emitted resolution proofs based on those axioms. The proofs could be checked by the resolution proof checker. The number of lines of code for that checker plus the number of lines for the LF axioms might still be smaller than 803 lines of C code. So this approach could support even more trusted checking of LF proofs. And for reasoning that can be represented directly as resolution proofs, the trusted computing base would be much smaller than with the LF checker.

Bibliography

[AF99] A. Appel and E. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communication Security*, 1999.

[AGM92] S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[AMSV02] A. Appel, N. Michael, A. Stump, and R. Virga. A trustworthy proof checker. In *Joint session of the FLoC '02 VERIFY and FCS workshops*, 2002.

[Bar92] H. Barendregt. *Lambda Calculi with Types*, pages 117–309. Volume 2 of Abramsky et al. [AGM92], 1992.

[Bar02] C. Barrett. *Checking Validity of Quantifier-free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, 2002. In preparation.

[BDL96] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, 1996.

[BDS00] C. Barrett, D. Dill, and A. Stump. A Framework for Cooperating Decision Procedures. In David McAllester, editor, *17th International Conference on Computer Aided Deduction*, volume 1831 of *LNAI*, pages 79–97. Springer-Verlag, 2000.

[BDS02a] C. Barrett, D. Dill, and A. Stump. A Generalization of Shostak's Method for Combining Decision Procedures. In *4th International Workshop on Frontiers of Combining Systems (FroCos)*, 2002.

[BDS02b] 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.

[Bee85] M. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.

[Bjo99] N. Bjorner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1999.

[BLM01] P. Bjesse, T. Leonard, and A. Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In G. Berry, H. Comon, and A. Finkel, editors, *13th Conference on Computer-Aided Verification*. Springer-Verlag, 2001.

[BN00] 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.

[BT97] F. Baader and C. Tinelli. A new approach for combining decision procedures for the word problem, and its connection to the Nelson-Oppen combination method. In W. McCune, editor, *14th International Conference on Computer Aided Deduction*, LNCS, pages 19–33. Springer-Verlag, 1997.

[BT00] L. Bachmair and A. Tiwari. Abstract Congruence Closure and Specializations. In David McAllester, editor, *17th International Conference on Automated Deduction*, volume 1831 of *LNAI*, pages 64–78. Springer-Verlag, 2000.

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1), 2001.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata, 1997.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA)*, 2001.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, 2001.

[CLR92] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combinations of Theories. In M. McRobbie and J. Slaney, editors, *13th International Conference on Computer Aided Deduction*, volume 1104 of *LNCS*, pages 463–477. Springer-Verlag, 1996.

[Coq91] T. Coquand. *An algorithm for testing conversion in Type Theory*, pages 255–79. In Huet and Plotkin [HP91], 1991.

[CS95] G. Collins and D. Syme. A Theory of Finite Maps. In *Conference on Higher Order Logic Theorem Proving and its Applications*, 1995.

[CtPg86] R. Constable and the PRL group. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[DS78] P. Downey and R. Sethi. Assignment Commands with Array References. *Journal of the ACM*, 25(4):652–666, October 1978.

[DV01] A. Degtyarev and A. Voronkov. *The Inverse Method*, chapter IV. In Robinson and Voronkov [RV01], 2001.

[E. 97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[End72] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[Fef95] S. Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995.

[FG00] W. Farmer and J. Guttman. A Set Theory with Support for Partial Functions. *Logica Studia*, 66(1):59–78, 2000.

[GK99] A. Geser and W. Kuchlin. Structure formal verification of a fragment of the IBM S/390 clock chip. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *International Workshop on Current Trends in Applied Formal Methods*. Springer-Verlag, 1999.

[GM92] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[HHP93] 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.

[Hod97] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[HP91] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.

[HP00] R. Harper and F. Pfenning. On Equivalence and Canonical Forms in the LF Type Theory. Technical Report CMU-CS-00-148, Carnegie Mellon University, July 2000.

[Jac99] B. Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.

[Kam] F. Kamareddine. Reviewing the classical and the de Bruijn notation for λ-calculus and pure type systems. *Logic and Computation*, 11(3):363–394.

[Kap68] D. Kaplan. Some Completeness Results in the Mathematical Theory of Computation. *Journal of the ACM*, 15(1):124–34, January 1968.

[KK94] M. Kerber and M. Kohlhase. A Mechanization of Strong Kleene Logic for Partial Functions. In A. Bundy, editor, *12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 371–385. Springer Verlag, 1994.

[KK97] M. Kerber and M. Kohlhase. Mechanising Partiality without Re-Implementation. In *21st Annual German Conference on Artificial Intelligence*, volume 1303 of *LNAI*, pages 123–134. Springer Verlag, 1997.

[Kle52] S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[KS00] W. Kuchlin and C. Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1-2):145–63, 2000.

[Lev99] J. Levitt. *Formal Verification Techniques for Digital Systems*. PhD thesis, Stanford University, 1999.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Edinburgh LFCS, 1992.

[McC62] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress 62*, 1962.

[Mit96] J. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Nec97] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[Nel81] G. Nelson. Techniques for Program Verification. Technical Report CSL-81-10, Xerox PARC, June 1981.

[NL98] G. Necula and P. Lee. Efficient representation and validation of proofs. In *13th Annual IEEE Symposium on Logic in Computer Science*, pages 93–104, 1998.

[NL00] G. Necula and P. Lee. Proof Generation in the Touchstone Theorem Prover. In David McAllester, editor, *17th International Conference on Automated Deduction*, 2000.

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, 1992.

[Pfe01] F. Pfenning. *Logical Frameworks*, chapter XXI. In Robinson and Voronkov [RV01], 2001.

[PS99] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[RS01] H. Ruess and N. Shankar. Deconstructing Shostak. In *16th IEEE Symposium on Logic in Computer Science*, 2001.

[Rue00] H. Ruess. Private communication. 2000.

[RV01] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.

[SBD02] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[SBDL01] A. Stump, C. Barrett, D. Dill, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Computer Society, 2001.

[SD02a] A. Stump and D. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *18th International Conference on Automated Deduction*, 2002.

[SD02b] A. Stump and D. Dill. Producing Proofs from an Arithmetic Decision Procedure in Elliptical LF. 2002. submitted to the 3rd International Workshop on Logical Frameworks and Meta-Languages.

[Sho84] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[SJ77] N. Suzuki and D. Jefferson. Verification Decidability of Presburger Array Programs. In *Proceedings of a Conference on Theoretical Computer Science*, 1977. University of Waterloo, Waterloo, Ontario, Canada.

[SJD98] J. Skakkebæk, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *10th International Conference on Computer Aided Verification*, 1998.

[Tar75] R. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215, 1975.

[TH96] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In F. Baader and K. Schulz, editors, *1st International Workshop on Frontiers of Combining Systems (FroCoS'96)*, volume 3 of *Applied Logic Series*. Kluwer, 1996.

[Vir99] R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, October 1999.

[Won99] W. Wong. Validation of HOL Proofs by Proof Checking. *Formal Methods in System Design*, 14(2):193–212, 1999.