Search Algorithms in Type Theory*

James L. Caldwell[†] Ian P. Gent[‡] Judith Underwood [§]

Revised submission to *Theoretical Computer Science*:

Special Issue on Proof Search in Type-theoretic Languages

May 1998

Abstract

In this paper, we take an abstract view of search by describing search procedures via particular kinds of proofs in type theory. We rely on the proofs-as-programs interpretation to extract programs from our proofs. Using these techniques we explore, in depth, a large family of search problems by parameterizing the specification of the problem. A constructive proof is presented which has as its computational content a correct search procedure for these problems. We show how a classical extension to an otherwise constructive system can be used to describe a typical use of the nonlocal control operator `call/cc`. Using the classical typing of nonlocal control we extend our purely constructive proof to incorporate a sophisticated backtracking technique known as 'conflict-directed backjumping' (CBJ). A variant of this proof is formalized in Nuprl yielding a correct-by-construction implementation of CBJ. The extracted program has been translated into Scheme and serves as the basis for an implementation of a new solution to the Hamiltonian circuit problem. This paper demonstrates a non-trivial application of the proofs-as-programs paradigm by applying the technique to the derivation of a sophisticated search algorithm; also, it shows the generality of the resulting implementation by demonstrating its application in a new problem domain for CBJ.

Keywords: Search Algorithms, Proofs-as-programs, Conflict-directed Backjump Search, Nonlocal Control (`call/cc`), Nuprl, Hamiltonian Circuit

*Authors are named in alphabetical order although Judith Underwood is principal author. We thank members of APES for their help and reviewers of this journal for their comments. Ian Gent and Judith Underwood especially thank Mr Denis Magnus for his invaluable contributions to their research.

[†]NASA Ames Research Center, Computational Sciences Division, Mail Stop 269-1, Moffett Field, CA, 94035-1000, USA. This work was performed while the author was visiting the Department of Computer Science, Cornell University, Ithaca, NY.

[‡]Department of Computer Science University of Strathclyde, Glasgow G1 1XH, United Kingdom. `ipg@cs.strath.ac.uk`

[§]Quality Systems and Software Ltd., 13 Atholl Crescent, Edinburgh, United Kingdom. `judith@qss.co.uk`. Judith Underwood was supported by EPSRC award GR/L/15685 while at the Department of Computing Science, University of Glasgow.

1 Introduction

Search problems are ubiquitous in Computer Science and Artificial Intelligence. For example, in Artificial Intelligence there has been extensive research into search algorithms for problems such as propositional satisfiability and constraint satisfaction problems. Powerful toolkits have been built, for example ILOG Solver, which has been applied to problems such as airport gate scheduling, timetabling, staff rostering, and manufacturing process scheduling [41]. Despite this success, there has been little recognition of the generality of most search techniques. Furthermore, many algorithms have been introduced without proof and only proved correct later, and typically with respect to, at best, pseudo-code. Some important algorithms have yet to be proved sound and complete. We address both these problems by showing how type theory can be used to prove search algorithms correct in a very general framework. As an example, we use 'Conflict-directed backjumping' (CBJ), which has never been verified in a formal framework. We outline a type-theoretic proof of this algorithm, report on a formal proof in Nuprl, the extraction of code for CBJ from this proof, and illustrate its use in the Hamiltonian Circuit problem, a domain in which CBJ has never previously been used.

Our approach to the verification of search algorithms uses type theory to connect the algorithm to a proof. We present a general description of theorems which, if proved constructively, induce search procedures. To verify a particular algorithm, we use it to guide development of a proof of an appropriate theorem. The computational content of that proof then constitutes a correct search procedure. This approach has been used to develop verified decision procedures, namely tableau proof search algorithms for classical and constructive propositional logic [10, 47]. In this paper, we extend the idea to a large class of problems involving search, to produce a general template for development of search procedures.

There are three main motivations of this work. First, we wished to create a framework which separates the search algorithm from domain specific information. This enables certain search techniques developed for specific problems to be applied in many more situations. For example, conflict-directed backjumping (CBJ) [40] was invented for solving scheduling problems, but can be applied to reduce search in a very wide range of problems. The original presentation of CBJ did not reflect its generality, which is revealed by our more abstract approach to search. In our framework, we can apply such techniques to new problems easily, producing implementations very quickly.

Second, we wished to reason about a typical use of nonlocal control, using a classical typing. Since the original discovery that Felleisen's control operator $\mathcal{C}$ could be given a type corresponding to the law of double negation elimination, a great deal of work has been done on the computational meaning of classical proof [2, 3, 6, 24, 33, 36]. However, these ideas have not been exploited in the context of program development or verification. To this end, we have shown how a limited use of classical reasoning in a proof can produce a program extraction which includes a nonlocal control operator. Furthermore, the control operator is used to return immediately from a (possibly deep) stack of recursive calls when a result is found. It was for such purposes that nonlocal control operators were added to purely functional languages, so this work demonstrates the practicality of the classical typing of nonlocal control for program development.

Finally, we wished to use new techniques for developing practical programs from proofs. This is the goal of much research in type theoretic theorem proving; however, comparatively little effort has been made to connect proof terms with programs recognizable to ordinary functional programmers. Recently, progress has been made in obtaining general recursive program (as opposed to strictly primitive recursive) as proof extracts [8]. Our work shows that these techniques can be applied to the development of a more substantial program with little difficulty.

The remainder of the paper is organized as follows:

- In section 2, we describe a general approach to search algorithms by considering how we may extract them from constructive proofs. We discuss the form of the theorem to be proved and outline a basic proof corresponding to a backtracking search procedure appropriate for many situations.

- In section 3, we consider the special case of a searching for an assignment of values to variables which satisfies some property. This case includes many search problems with significant practical applications. The theorem, proof, and supporting data structures and lemmas required for this situation are presented in some detail.

- In section 4, we present ideas which may be used to reduce search and describe what we need to do to incorporate these ideas in the proof. In particular, in section 4.2, we present conflict-directed backjumping from a logical point of view.

- In section 5, we extend the proof of section 3 with the search reduction techniques of section 4. In this proof, we use classical logic in a restricted way in order to obtain an implementation of backjumping which uses nonlocal control.

- In section 6, we describe the formalization in Nuprl, and the extraction of the corresponding program. We consider how particular features of Nuprl can be used to obtain an extract resembling a typical functional program.

- In section 7, we describe the use of this program to implement conflict-directed backjumping for the Hamiltonian circuit problem, a domain in which it has not been described before. We show that the technique is able to reduce search significantly on many test instances.

- Finally, in section 8 we discuss some generalizations and future work.

2 General description of search

In this section, we present a framework which describes many search problems and algorithms for solving them. In general, a search problem is a search for a structure satisfying some property P; for example, searching for a proof of a given formula is a search problem. An algorithm exists for this problem only if it is decidable whether or not there exists a structure satisfying P. We show that a constructive proof of this fact can have a natural search algorithm as its computational content.

In our framework, the details of the structure are largely unimportant, as long as only finitely many possible structures need to be searched. Similarly, as long as it is decidable whether or not a structure has property P, we are not concerned with the details of P. Hence, although a particular search problem will only be an instance of a general class of problems, the algorithm will be readily generalizable if P can be parametrized. For example, a search problem might be to search for a proof of a particular formula; however, the specification of what constitutes a proof of a formula can be parametrized by the formula, so that the algorithm can be used for general proof search.

Given a definition of the structure involved, and given the decidable predicate P on these structures, the general shape of the theorem we wish to prove is

$$(\exists s : \text{Structure}.\, P(s)) \vee (\forall s : \text{Structure}.\, \neg P(s)).$$

To guarantee correctness of the resulting algorithm, we must know that if the search for a structure satisfying P fails, then there is in fact no structure satisfying P. In proof search, a concise way of proving that no proof of a formula exists is to use model theory. Given a model theory for a logic and a constructive soundness proof, then a model in which the formula does not hold suffices to show that there is no proof of the formula. We thus extend the statement of the theorem to make this kind of reasoning explicit, without relying on a particular notion of a model theory:

$$(\exists s : \text{Structure}.\, P(s)) \vee (\exists R : \text{Reason}.\, R \to \forall s : \text{Structure}.\, \neg P(s))$$

Note that in the context of proof search, if R is a model in some sound model theory for the logic, then a proof of this theorem is, in fact, a completeness proof for the logic [13, 46]. We will see (in section 4.2) that we may be able to exploit R to prune the search space and thus generate more efficient search algorithms.

We now consider further the design of the proof and the algorithm we expect to derive from it. The proof will be such that the computational content is a backtracking search procedure. A structure is developed in stages, with each stage consisting of an extension to a partial structure, until a complete structure is built and can be tested for the property P. If no extension to a partial structure satisfies P, then the last choice of extension is undone and another choice tried until all choices are exhausted. Thus, for each partial structure there must be finitely many choices for extension, and the process of extension must be known to terminate. We capture these conditions in a requirement that there be a well-founded measure associated with the relation p extends ps. Such measures may be quite complex; however, the proof relies only on the existence of such a measure to ensure that induction is valid. The computational content of the proof need not refer to the measure at all.

Using this induction measure, we prove the theorem by proving a more general theorem in which we can exploit the inductive hypothesis. The statement of this theorem is:

$$\begin{array}{l}\forall ps : \text{Partial Structure.} \\ (\exists s : \text{Structure.}\, s \text{ extends } ps \wedge P(s)) \\ \vee \\ (\exists R : \text{Reason.}\, R \rightarrow \forall s : \text{Structure.}\, s \text{ extends } ps \rightarrow \neg P(s))\end{array}$$

Given a partial structure ps, we can extend it and then apply the inductive hypothesis on the extended structure. This corresponds to a recursive call to the function representing the computational content of the structure. If multiple extensions are possible, these must be accounted for in some way by the inductive measure, though in the next section we will see how this can be done without predetermining the choice of extension. The base case of the induction occurs when no extension is possible; since we have assumed P is decidable, this case is proved by reference to P.

As before, we must have evidence for the negative case, in the form of R which implies that no extension of ps satisfies P. The evidence R will generally depend on ps, and must cover all the possible structures which extend ps. Careful choice of the type of R and analysis of a given R can lead to more efficient search procedures. In particular, if we are given a partial structure ps and evidence R that no extension of ps satisfies P, then it may happen that R is also evidence that no extension of ps' satisfies P, where ps is an extension of ps'. Then, instead of backtracking one step from ps, we may wish to *backjump* to the point where ps' was constructed, which may be much earlier in the search.

Backjumping as a search reduction technique was first described by Gaschnig [20], but his presentation was very limited in the amount of backjumping performed. Conflict-directed backjumping (CBJ), a more extensive form of backjumping, was first described by Prosser in the context of a scheduling problem [39]. Later, it was generalized to binary constraint satisfaction problems [40], and arbitrary constraint satisfaction problems [22]. Here, we have generalized it further and view it as a general search reduction technique rather than as an algorithm for a particular problem. Indeed, in this paper we give a proof whose computational content allows CBJ to be implemented for any problem which can be formulated with a finite set of variables taking a finite set of values, and where we search for an assignment which satisfies some decidable predicate. Among examples of such problems are all NP-complete problems [19], although we make no assumptions which limit us to NP problems. Informal proofs of correctness of CBJ have been presented by Ginsberg [22] and Kondrak and van Beek [29], but until now it has not been proved correct in a formal framework. The value of CBJ as a search reduction technique has been shown experimentally [43, 23, 1, 5] and theoretically [29], and we show for the first time that the technique can reduce search in the Hamiltonian circuit problem.

We have given a very general logical description of how some theorems can be proved in such a way as to generate search procedures. To illustrate this general pattern, in the next section we will see a more concrete example. This example fixes the structure for which we are searching, but still describes a large family of search problems.

3 Search for assignments of values to variables

In this section we consider the case in which the structure for which we are searching is an assignment of values to variables. Specifically, we assume we have a finite set *Varset* of variables, a finite set *Valset* of values, and a decidable predicate P on assignments of values in *Valset* to the variables in *Varset*.

We explore this special case for many reasons. Although we have made the structure explicit, we can express a large number of search problems as predicates on assignments: constraint satisfaction, Boolean satisfiability, and other NP-complete problems. Search algorithms for these problems have compelling practical interest, and many techniques have been introduced in an effort to improve efficiency of search in this area.

However, improvements to general search algorithms are rarely presented as such in the literature; instead, a method is often introduced in one problem area (e.g. constraint satisfaction) and later applied to other areas. For example, conflict-directed backjumping was described for constraint satisfaction problems [40, 22] some years before being applied successfully to propositional satisfiability [4, 5] and it has not previously been presented as generally as we do so here. Our approach separates the properties of the search algorithm from the problem specific functions, permitting the core search algorithm to be applied to many different domains by varying the predicate P. In developing the proof of the theorem from which the search algorithm will be extracted, we will generate correctness conditions for the problem specific routines on which the final program will depend.

In this case, the high level theorem we will prove is

Theorem 1 *Given a finite set of variables, Varset, a finite set of values Valset, and a predicate P on assignments of variables to values, then*

$$(\exists A : Assign(Varset, Valset).\, P(A)) \vee (\forall A : Assign(Varset, Valset).\, \neg P(A))$$

In this formulation, the structure desired is an assignment, rather than a proof, and the alternative is a proof that no assignment exists. While an assignment is a proof that a given problem has a solution, this is not the normal interpretation of proof. This might, then, appear to be a shift in interpretation of the framework outlined in the previous section, if "structure" was interpreted as 'proof' and "reason" interpreted as 'counter-model'. Such an interpretation is valid in many examples of proof search, say in resolution proof search where a full proof is often easily extracted when found. But there are many examples of proof search where we are not interested in the proof object as such, merely that one exists and thus that the given conjecture is a theorem. For example, proof systems based on semantic tableaux [45] typically search for a counter-model: if a counter-model is found, it is produced as evidence that the conjecture is false, while if none is found the constructed tableau is often discarded. To do otherwise, even in the case of propositional logic, would involve the storage of an exponentially large proof. This is the point of view we use here. It is particularly appropriate in the case of search in NP-complete problems where the satisfying assignment, if it exists, might represent a valuable schedule, timetable, or plan, while the details of the proof that no such structure exists may be of no interest to the user. Nevertheless, the notion of a 'reason' for no structure existing plays a vital part in our work, and we use it to provide evidence to skip over parts of the search space in backjumping. Applying our methodology to reduce the amount of search for explicit proof objects remains future work, as we discuss in section 8.

The initial proof of Theorem 1 will result in a core search algorithm performing a simple recursive backtracking search procedure. At each stage we will have a partial assignment of values to variables. To advance the search, an unassigned variable is chosen and given one of its possible values to create a new assignment on which the search procedure is called recursively. If this new assignment cannot be extended to an assignment with the desired property, another value for the variable is chosen until all values have been tried. If all values fail, then the procedure returns to try a new value for a previously assigned variable. Thus, the procedure begins with an empty assignment and builds an assignment in stages. An assignment can only be extended finitely often, since we assume we begin with a finite set of variables, and only finitely many values are possible.

This basic backtracking search algorithm will be later used as a base for extension, but for now we will use it as a guide for the proof of Theorem 1. To clarify both the proof and the resulting program, we treat separately the extension of an assignment by the choice of a new variable to set, and the process of trying a new value for a given variable. Thus, the proof of the theorem is by induction on the set of variables yet to be assigned. After one such variable is chosen, we have another induction on the set of values yet to be tried for that variable. These could be packaged together in a single induction principle; the separation is only for ease of understanding. After choosing a value for the variable, we create an extended partial assignment and use the first inductive hypothesis, since this new assignment has fewer unassigned variables. Then, if necessary, we apply the second inductive hypothesis, since we have reduced the number of values yet to be tried. The use of an inductive hypothesis in the proof corresponds to a recursive call in the algorithm.

We now present the ideas used in the statement and proof of Theorem 1, followed by an outline of the proof. We will then describe extensions to the proof which permit the algorithm to include various heuristics and search pruning techniques, and in section 5 we describe the extended proof in some detail. A discussion of the issues arising from the formalization of the proof in Nuprl can be found in section 6.

3.1 Notation, definitions, and assumptions

We begin by describing in more detail the types involved in the statement of the theorem and the lemmas which will be used in the proof. The description of the data-types constitutes an informal specification of their intended behavior. The program which results is thus correct if we are given a correct implementation of the data-types. Similarly, the lemmas which we assume will have computational meaning, and we describe how their proofs will affect the final algorithm. The lemmas will be named, and occasionally we will refer to "calling" a lemma when we wish to use the computational content of a lemma.

Finite sets. We assume we have a type of finite sets, and, in particular, a finite set *Varset* of variables and a finite set *Valset* of values. We suppose that all the standard set operations are defined, e.g. union, member, remove, and subset relations. We also assume we have an induction principle on finite sets ordered by inclusion. The computational content of the induction principle should be a mechanism for defining functions inductively; this is described further in section 6 below.

Assignments. Given *Varset* and *Valset*, we assume we have a type of assignments, $Assign(Varset, Valset)$ from the variables in *Varset* to the values in *Valset*. We consider that this type includes partial assignments defined only on subsets of *Varset*. To emphasize that we are dealing with such a partial assignment, we may write $A : Assign(Vars, Valset)$ to describe that A is defined on *at most* the set of variables *Vars*. We also assume we have a type $Full\text{-}Assign(Varset, Valset)$, a subtype of $Assign(Varset, Valset)$ describing total assignments on *Varset*. We assume we have a strict and non-strict partial order on assignments, denoted $\subset$ and $\subseteq$ respectively, which describe when one assignment extends another by determining values for at least as many variables.

Predicates and conflict sets. We assume P is a decidable predicate on the type $Full\text{-}Assign(Varset, Valset)$. To describe the case that no extension of a partial assignment satisfies P, we introduce the notion of a *conflict set*. Given a partial assignment $A : Assign(Vars, Valset)$, a conflict set for A is a subset CS of *Vars* such that

$$\begin{array}{l}\forall A' : Full\text{-}Assign(Varset, Valset).\\ \quad P(A') \rightarrow\\ \quad\quad \exists v_0 \in CS.\ Val\_of\,(v_0, A') \neq Val\_of\,(v_0, A)\end{array}$$

Such a set CS serves as evidence that A cannot be extended to an assignment satisfying P, since any assignment satisfying P differs from A on the value of some variable v_0 in the conflict set. This idea will be exploited further in the next section. In the base proof, we assume we have the following lemma:

Lemma 2 (`check-full`)

$$
\begin{array}{l}
\forall A : \textit{Full-Assign}(\textit{Varset}, \textit{Valset}). \\
\quad P(A) \\
\quad \vee \\
\quad \exists CS \subset \textit{Varset}. \\
\quad\quad \forall A' : \textit{Full-Assign}(\textit{Varset}, \textit{Valset}). \\
\quad\quad\quad P(A') \rightarrow \exists v_0 \in CS.\ \textit{Val\_of}\,(v_0, A') \neq \textit{Val\_of}\,(v_0, A)
\end{array}
$$

The computational content of this lemma extends the decision procedure for P to produce either a proof of $P(A)$ or a conflict set as evidence of $\neg P(A)$. Note that if A : *Full-Assign*(*Varset*, *Valset*) does not satisfy P, then the set *Varset* is always a conflict set. Conflict sets play an important role in the proof and corresponding algorithm; this will be further explored in section 4.2.

Choosing an element of a set. In the course of the proof, we shall need to choose elements of given nonempty sets of variable and values. To do this, we assume the following lemmas:

Lemma 3 (`choose-var`)

$$\forall V : \{\textit{Vars} \subseteq \textit{Varset} | \textit{Vars} \neq \emptyset\}. \exists v \in V$$

Lemma 4 (`choose-val`)

$$\forall V : \{\textit{Vals} \subseteq \textit{Valset} | \textit{Vals} \neq \emptyset\}. \exists n \in V$$

The computational content of the proofs of these lemmas serve as functions which choose the next variable to set, and the next value to try. Different proofs for these lemmas will, in general, result in different algorithms, but since the main theorem assumes nothing about the proofs of these lemmas, all of the resulting algorithms will be correct.

3.2 Core proof outline

We prove the main theorem above as a corollary to the following more general theorem.

Theorem 5 (`test`) *Given a finite set of variables, Varset, a finite set of values Valset, and a predicate P on full assignments of values to variables, then*

$$
\begin{array}{l}
\forall \textit{Vars} \subseteq \textit{Varset}. \\
\quad \forall A : \textit{Assign}(\textit{Vars}, \textit{Valset}). \\
\quad\quad \exists A' : \textit{Full-Assign}(\textit{Varset}, \textit{Valset}).\ A \subseteq A' \wedge P(A') \\
\quad\quad \vee \\
\quad\quad \exists CS \subseteq \textit{Vars}. \forall A' : \textit{Full-Assign}(\textit{Varset}, \textit{Valset}). \\
\quad\quad\quad P(A') \rightarrow \exists v_0 \in CS.\ \textit{Val\_of}\,(v_0, A') \neq \textit{Val\_of}\,(v_0, A)
\end{array}
$$

Theorem 1 follows from this theorem by taking *Vars*, the set of assigned variables, to be empty, and taking A to be the empty assignment. ¿From this we have either A' satisfying P or a conflict set showing that the empty assignment cannot be extended to as assignment satisfying P. From such a conflict set, we can show $\forall A : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\, \neg P(A)$ as desired.

Before we begin the proof of Theorem 5, we define an abbreviation. If $A : \mathit{Assign}(\mathit{Vars}, \mathit{Valset})$, let $\mathit{Result}(A)$ be

$$\begin{array}{l} (\exists A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\, A \subseteq A' \wedge P(A')) \\ \vee \quad (\exists CS \subseteq \mathit{Vars}.\, \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\ \qquad P(A') \rightarrow \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A)) \end{array}$$

$\mathit{Result}(A)$ denotes both the formula above and the type of the result of applying the search procedure to the partial assignment A – it returns either a full assignment extending A and satisfying P, or it returns a conflict set.

The proof of Theorem 5 is by induction on the size of the set $\mathit{Varset} - \mathit{Vars}$, the set of variables which have not yet been assigned values. More precisely, it is on the finite sets of unassigned values ordered by subset. We denote such sets *Varsleft*, and so we will generally have $\mathit{Vars} = \mathit{Varset} - \mathit{Varsleft}$, i.e. the set of assigned variables is the difference between the set of variables and the set of unassigned variables. The base case is when $\mathit{Varsleft} (= \mathit{Varset} - \mathit{Vars})$ is empty. Then the assignment A assigns a value to every variable. We can then use Lemma `check-full` to prove the result.

For the inductive case of this first induction, we assume *Varsleft* is a nonempty set of unset variables. We have the following as an inductive hypothesis:

$$IH1 : \forall s \subset \mathit{Varsleft}.\, \forall A' : \mathit{Assign}(\mathit{Varset} - s, \mathit{Valset}).\, \mathit{Result}(A')$$

and we must prove

$$\forall A : \mathit{Assign}(\mathit{Varset} - \mathit{Varsleft}, \mathit{Valset}).\, \mathit{Result}(A)$$

In other words, given an assignment $A : \mathit{Assign}(\mathit{Varset} - \mathit{Varsleft}, \mathit{Valset})$ we need to construct either a satisfying assignment extending A or a conflict set for A.

We construct $\mathit{Result}(A)$ by creating an extension of A and then using the inductive hypothesis. First, we choose a variable in *Varsleft*, using Lemma `choose-var`. Given this variable, we create a sequence of extensions by trying all the possible values for it. We do this by using the following lemma:

Lemma 6 (`enumerate-domain`)
Given an assignment $A : \mathit{Assign}(\mathit{Varset} - \mathit{Varsleft}, \mathit{Valset})$ *and* $v \in \mathit{Varsleft}$,

$$\begin{array}{l} \forall \mathit{Vals} \subseteq \mathit{Valset}. \\ \quad \exists A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\, A \subseteq A' \wedge P(A') \wedge \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \\ \vee \\ \quad \exists CS \subseteq \mathit{Varset} - \mathit{Varsleft}.\, \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\ \qquad \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \rightarrow P(A') \rightarrow \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A) \end{array}$$

Note that the specification of CS here is slightly different from that described above. There are really two kinds of conflict sets: one which serves as evidence that a partial assignment A cannot be extended to one satisfying P, and one which is evidence that a particular collection of partial assignments extending A cannot be extended to assignments satisfying P. This distinction is discussed in greater detail in section 5.

Lemma 6 is proved by induction on the size of the set *Vals*. Given the lemma, we can prove *Result*(A) (and hence complete the proof of Theorem 5) by applying the lemma with *Vals* = *Valset*. It is important to note that this lemma is proved in the context of the proof of `test`, as we will use $IH1$ in the proof. Of course, the lemma could also be stated separately as long as its hypotheses included $IH1$ and the other elements of the current context used.

The base case of the induction is when *Vals* is empty. There are no full assignments giving v a value in the empty set, so in particular there is no full assignment extending A which gives v a value in the empty set. Thus, we must create a conflict set CS. However, the property which CS must satisfy is trivial, since $\mathit{Val\_of}\,(v, A') \in \mathit{Vals}$ will always be false. Thus, any conflict set will do; for example, the empty set.

In the inductive case, we have a second inductive hypothesis:

$$
\begin{array}{l}
IH2(Vals) : \forall vs \subset \mathit{Vals}. \\
\quad \exists A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\, A \subseteq A' \wedge P(A') \wedge \mathit{Val\_of}\,(v, A') \in vs \\
\quad \vee \\
\quad \exists CS \subseteq \mathit{Varset} - \mathit{Varsleft}. \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\
\qquad \mathit{Val\_of}\,(v, A') \in vs \rightarrow P(A') \rightarrow \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A)
\end{array}
$$

We then wish to prove

$$
\begin{array}{l}
\exists A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\, A \subseteq A' \wedge P(A') \wedge \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \\
\vee \\
\exists CS \subseteq \mathit{Varset} - \mathit{Varsleft}. \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\
\quad \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \rightarrow P(A') \rightarrow \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A)
\end{array}
$$

Here, *Vals* represents the set of values which have yet to be tried as values of the variable v. Thus, to apply the second inductive hypothesis, we must reduce this set. So choose a value n in *Vals*, using Lemma `choose-val`. Let $A_{v=n}$ be the assignment A extended with v equal to n.

Now we must try the partial assignments extending $A_{v=n}$. Since $A \subset A_{v=n}$, we do this by applying the first inductive hypothesis. Computationally, this corresponds to a recursive call to `test`. The inductive hypothesis provides something of type *Result*($A_{v=n}$); that is, it is either an assignment A' extending $A_{v=n}$ such that $P(A')$, or it is a conflict set for $A_{v=n}$, as described above. If we have a solution A', we are done.

If not, then we have a conflict set (call it $CS1$) for $A_{v=n}$. Now we remove n from *Vals* and apply the second inductive hypothesis (computationally, a recursive call to `enumerate-domain`) with the set $\mathit{Vals} - \{n\}$. If the result is an

A' : *Full-Assign*(*Varset*, *Valset*) such that $A \subseteq A'$ and $P(A')$, then we are done. Otherwise, we have a second conflict set $CS2 \subseteq \mathit{Varset} - \mathit{Varsleft}$ satisfying

$$\forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\,(\mathit{Val\_of}\,(v, A') \in \mathit{Vals} - \{n\}) \rightarrow \\ P(A') \rightarrow \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A)$$

Now let $CS = (CS1 - \{v\}) \cup CS2$. Then $CS \subseteq \mathit{Varset} - \mathit{Varsleft}$, and it is easy to check that CS satisfies

$$\forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}).\,(\mathit{Val\_of}\,(v, A') \in \mathit{Vals}) \rightarrow \\ P(A') \rightarrow \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A)$$

Thus we have finished the inductive case of the lemma, and hence we have finished the proof. Next, we consider some modifications of the proof resulting in more efficient algorithms.

4 Extending the search procedure

The computational content of the proof outlined in the preceding section is a simple backtracking search procedure. The actual search performed depends on the computational content of the proofs of Lemmas `choose-var` and `choose-val`. Thus, it is easy to incorporate variable and value ordering heuristics in the proof, simply by choosing appropriate proofs of these lemmas.

Other search optimization techniques can be incorporated by modifications to the proof. In this section, we describe two of these optimization techniques in detail. The first technique is a simple check on the consistency of a partial assignment. Second, we describe conflict-directed backjumping [40] and how it may be implemented by a modification to the proof. The approach we take is particularly interesting since it permits the computational extract to use a nonlocal control operator to perform backjumping.

4.1 Consistency checking

The simplest extension to the proof which we consider permits failure to occur before a full assignment has been created. It is common that a partial assignment will already contain enough information to determine the falsity of the property P for any extension. For example, in clause form satisfiability problems, a partial assignment which falsifies one clause cannot be extended to a satisfying assignment. Thus, we do not need to explore this part of the search space further.

To add this consistency check to the proof (and hence to the algorithm), we need only assume the following lemma:

Lemma 7 (`check`)

$$
\begin{array}{l}
\forall A : \mathit{Assign}(\mathit{Vars}, \mathit{Valset}). \\
\quad \mathit{unit} \\
\quad \vee \\
\quad \exists CS \subset \mathit{Varset}. \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\
\qquad P(A') \to \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A)
\end{array}
$$

Given a partial assignment A, this lemma either provides evidence that A cannot be extended to an assignment satisfying P, or it returns a token (represented here as an element of unit, a type with one element) which signifies that search should continue. Note that this lemma is always provable since we may simply always return an element of unit, though such a proof of the lemma would add nothing to the original algorithm. However, if the partial assignment is inconsistent, evidence for this must be provided in the form of a conflict set.

This lemma is applied in the proof when the new assignment $A_{v=n}$ is created. At that point in the proof above, we applied the first inductive hypothesis to obtain a proof of $\mathit{Result}(A_{v=n})$. Computationally, this step corresponds to a recursive call of the search algorithm on the new assignment, and involves a search of all extensions of $A_{v=n}$. In the modified proof, we first apply Lemma `check` to see if we can find evidence for inconsistency immediately. If we produce a conflict set CS for $A_{v=n}$, then we can proceed without appealing to the inductive hypothesis. We have that $A \subseteq A'$, and if the inclusion is strict there is no need to check all the other full assignments extending A. This could save a considerable amount of search in the resulting search algorithm.

4.2 Conflict-directed backjumping

The second technique we consider is more complex. Conflict-directed backjumping is a means of using the information in a conflict set to reduce search. When a conflict set CS for a partial assignment A is produced, it satisfies

$$
\begin{array}{l}
\forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\
\quad P(A') \to \exists v_0 \in CS.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A).
\end{array}
$$

Now, if CS is smaller than the set of variables assigned in A, it may be that CS satisfies the same condition for a partial assignment A'' such that A was created by extending A'', possibly many times. In other words, CS might be evidence that some much smaller assignment A'' cannot be extended to an assignment satisfying P. Rather than continue to explore assignments which extend A'', we wish to return search to the point at which A'' was created, using the conflict set CS as evidence of failure. We now have that $A'' \subseteq A \subseteq A'$, and if the first inclusion is strict, there is no need to explore all the partial assignments extending A''. Just as with the use of `check`, this may save a lot of needless search in the resulting algorithm.

To implement this idea, we need a means of returning a value to an earlier point in the search tree without having explored the entire tree below that point. This can be done by adding a test whenever a conflict set is returned, and deciding whether to pass it back or to continue search below the given assignment. However, in this development we use a different technique, that of explicit management of the continuation through use of call/cc.

The call/cc (or *call-with-current-continuation*) operator was introduced to the Scheme programming language [12] to permit direct manipulation of program control. When call/cc$(\lambda(k)\ldots)$ is evaluated, k becomes bound to the current continuation; in other words, k represents the rest of the computation, apart from that remaining in the body of the call/cc. When k is applied to an argument, the computation returns immediately to the context which existed when k was created, and the argument passed to k is used in the place of the call/cc$(\lambda(k)\ldots)$ term. Thus, call/cc is essentially a functional goto; it allows control to jump immediately to another part of the program. Typical uses of call/cc include error handling and implementation of coroutines [18, 17, 16]. We can view the use of call/cc to implement backjumping as a form of error handling, allowing immediate return to the point at which a decision was made to explore a branch now known not to contain a solution.

In the program extracted from the proof, we wish to use call/cc to create continuations which represent points to which the search might backjump. Backjumping occurs when a conflict set is found which eliminates more of the search tree than its local situation requires. A continuation is created whenever a variable is set to create a partial assignment. Should we discover, deep in the search tree, that this partial assignment is inconsistent, we return immediately to this point by applying the continuation to the evidence of inconsistency, in the form of a conflict set.

We may be deep in the search tree when we discover that no assignment with the current values of two early variables is possible. We wish to jump back immediately and try another value for the second variable. For example, suppose that we have a problem involving propositional Boolean variables a to z and that the problem is to find an assignment of these variable to Boolean values satisfying $(\neg a \vee z) \wedge (\neg c \vee \neg z)$ If we consider variables in alphabetical order, and T as a value before F, the first assignment checked would be $a = T, b = T, c = T, \ldots y = T, z = T$, but this would fail with conflict set $\{c, z\}$ because of the second clause. The next assignment considered would be $a = T, b = T, c = T, \ldots y = T, z = F$, but this would also fail, with conflict set $\{a, z\}$ because of the first clause. Therefore no solution is possible with the partial assignment $a = T, b = T, c = T, \ldots y = T$. But there is no point in backtracking through other values of variables $d, e \ldots y$; the value of either a or c must be changed. The conflict set is $\{a, c\}$. Search may return immediately to try a new value for c. We can do this by applying the continuation created at the time variable c was being set to the evidence (in the form of a conflict set) that the assignment is inconsistent. In this case we would try the value $c = F$ and would succeed with any assignment extending this. If however other clauses were present which ruled out $c = F$ without involving variable b, we could backjump over b to try

$a = F$. The success of backjumping techniques is not limited to such contrived examples, and has been shown in larger problems [40, 43, 1, 5].

To get this computational behavior from the proof, we use the fact that call/cc can be given the type $((\alpha \to \beta) \to \alpha) \to \alpha$ for any types α and β [24, 26]. This is a classical axiom which corresponds to a form of proof by contradiction, particularly when we take β to be $\perp$, or falsity. If, from the assumption that α implies false, we can prove α, then we have a contradiction so α must be true. This form of reasoning is not strictly constructive, but in this case we still have a computational meaning for it. Although a constructive formal system like Nuprl does not permit classical reasoning, we can add it by adding to the theorem the assumption that call/cc has type

$$\forall\alpha\forall\beta.\, ((\alpha \to \beta) \to \alpha) \to \alpha.$$

Of course, Nuprl does not describe the computational behavior of call/cc, so we must justify this typing theoretically. The observation that nonlocal control operators could be given classical types is due to Griffin [24]; Murthy [33, 34] developed this idea to describe in more detail the connection between continuation passing style program transformations and translations of classical logic into constructive logic. There are limits of program extraction from classical proof [33]; however, our use of this connection is justifiable using this work.

To accommodate these ideas, we must modify the statement and proof of the theorem. In the statement of the theorem, we add an extra assumption of the form $\forall A_0 \subseteq A.\, (\mathit{Result}(A_0) \to \perp)$. This assumption is satisfied by a function which uses continuations created by call/cc. When we produce a more general conflict set than is required and wish to backjump, we apply the function, which then uses the appropriate continuation to return immediately to the right stage in the computation. Logically, this step is an unnecessarily roundabout proof of $\mathit{Result}(A)$. If the conflict set CS is really a valid result for some previous partial assignment A_0, then we use the assumption $\neg\mathit{Result}(A_0)$ to get a contradiction and hence to conclude anything, and in particular $\mathit{Result}(A)$. However, when the continuation corresponding to the assumption $\neg\mathit{Result}(A_0)$ is applied, the computation returns to the point where A_0 has been created and is being tested. The conflict set CS is now treated as a conflict set for A_0, and computation continues from that point.

This logical treatment of the nonlocal control ensures that backjumping is sound; we can only backjump when we have evidence that there is no solution in the part of the search tree we are pruning. Naturally, the proof corresponding to a backjumping algorithm is more complex than the proof corresponding to a simple backtracking algorithm; since the program is more complex as well this should not be surprising. What is perhaps surprising is that the modifications necessary are not even more complex.

5 Details of the extended proof

In this section, we describe in some detail the proof including both consistency checking and conflict-directed backjumping. Adding a consistency check is relatively simple; logically, it corresponds to an appeal to a lemma which may or may not prove the current goal. Adding conflict-directed backjumping requires a modification of the statement of the main theorem, Theorem 5, and a few extra steps in its proof.

To introduce the ideas used in the proof, we will take a closer look at conflict sets and their use. In the base proof, conflict sets served only as evidence that no extension of a partial assignment satisfied the property P. In the conflict-directed backjumping algorithm, the elements of a conflict set are used to guide the search directly. Given a conflict set, backjumping returns search as far as possible while still preserving the property that the conflict set serves as evidence that the partial assignment cannot be extended.

How far back the search goes depends on the variables in the conflict set. In any conflict set, some variable is "most recently set". The search algorithm should return to the point at which this variable was set and try a new value for it. In order to return using call/cc, we must have captured the continuation at this point earlier in the computation. The desired point to which we expect control to return will guide our use of classical logic (and thus call/cc) in the proof.

The search algorithm does not simply return control; it returns a conflict set which serves as evidence that no solution exists in part of the search tree. In fact, the role of the conflict set changes subtly when backjumping occurs. Before, it is evidence that a given partial assignment cannot be extended to satisfy P. After, it is evidence that a partial assignment cannot be extended to satisfy P in such a way that the current variable has one of a given set of values.

More precisely, suppose we have a partial assignment A_i defined on the variables $\{v_1, \ldots v_i\}$. We attempt to find an assignment satisfying P by extending this assignment with $v_{i+1} = n_1$, creating an assignment A_{i+1}. Suppose that further along this branch of the search tree, we have an extension A_m of A_i defined on $\{v_1, \ldots, v_i, v_{i+1}, \ldots, v_m\}$. Also, suppose we have discovered that A_m cannot be extended to an assignment satisfying P. The conflict set CS_m which serves as evidence for this statement must satisfy the following predicate:

$$\begin{aligned}&\forall A' : \textit{Full-Assign}(\textit{Varset}, \textit{Valset}).\, P(A') \rightarrow \\ &\quad \exists v_0 \in CS_m.\ \textit{Val\_of}\,(v_0, A') \neq \textit{Val\_of}\,(v_0, A_m)\end{aligned}$$

Now suppose that the most recently set variable in CS_m is v_{i+1} – in other words, $CS_m \subseteq \{v_1, \ldots, v_{i+1}\}$ and $v_{i+1} \in CS_m$. Then CS_m serves as a conflict set for the partial assignment A_{i+1} as well, but not for any smaller partial assignment. To use this fact, we now want search to return to the point at which variable v_{i+1} was assigned the value n_1, and try a new value for this variable. Note that CS_m is not a conflict set for A_i; it is evidence that setting $v_{i+1} = n_1$ failed, and may be used to build a conflict set for A_i if all other attempts to extend A_i fail. However, a conflict set for A_i cannot contain the variable v_{i+1}. So, consider the set $CS_i = CS_m - v_{i+1}$. CS_i is then a subset of $\{v_1, \ldots, v_i\}$ and also satisfies

$$\begin{array}{l} \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\ \quad \mathit{Val\_of}\,(v_{i+1}, A') \in \{n_1\} \rightarrow P(A') \rightarrow \\ \quad\; \exists v_0 \in CS_i.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A) \end{array}$$

As further extensions to A_i are explored by setting v_{i+1} to $n_2, \ldots, n_j$ in turn, as long as no solution is found we keep a set $CS'_i \subseteq \{v_1, \ldots, v_i\}$ which satisfies

$$\begin{array}{l} \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\ \quad \mathit{Val\_of}\,(v_{i+1}, A') \in \{n_1, \ldots, n_j\} \rightarrow P(A') \rightarrow \\ \quad\; \exists v_0 \in CS'_i.\ \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A) \end{array}$$

Thus, there are two ways of viewing a conflict set: as evidence that a partial assignment cannot be extended to a solution, and as evidence that a partial assignment cannot be extended to a solution if the next variable has a value in a specified set. Another way of looking at this second kind of conflict set is that it shows that a family of partial assignments cannot be extended. Before backjumping, the conflict set built is evidence of the first kind. After, it is evidence of the second kind, with respect to a smaller partial assignment. The link between these two views of the conflict set is described in part by the treatment of the continuation function, which, when backjumping, transforms the type of the conflict set.

The backjumping algorithm must determine what the most recently set variable in the conflict set actually is. This intended behavior determines how the continuation created by call/cc is actually used in the proof, and hence in the program. The control operator call/cc used to create a continuation which performs the backjumping; the application of this continuation occurs inside what we will call a *continuation function*, which determines the point to which the algorithm backjumps, the conflict set it returns, and applies the appropriate continuation.

With these ideas in mind, we reconsider the proof of Theorem `test`. The revised statement of the theorem is:

Theorem 8 (classical test) *Given a finite set of variables, Varset, a finite set of values, Valset, and a predicate P on full assignments of values to variables, then*

$$\begin{array}{l} \forall\, \mathit{Vars} \subseteq \mathit{Varset}. \\ \quad \forall A : \mathit{Assign}(\mathit{Vars}, \mathit{Valset}). \\ \quad\quad (\forall A_0 : \mathit{Assign}(\mathit{Vars}, \mathit{Valset}).\, A_0 \subseteq A \rightarrow (\mathit{Result}(A_0) \rightarrow \bot)) \rightarrow \\ \quad\quad \mathit{Result}(A) \end{array}$$

Before we prove this theorem, we show how it is used to prove the top level theorem, Theorem 1. To prove Theorem 1, we prove instead $\mathit{Result}(A_{empty})$, the result for the empty assignment. With this as our goal, we use call/cc with $\alpha = \mathit{Result}(A_{empty})$ and $\beta = \bot$ to introduce the following assumption,

$Result(A_{empty}) \to \bot$. We then appeal to Theorem 8, taking *Vars* to be empty, A to be the empty assignment, and using the assumption $Result(A_{empty}) \to \bot$ to obtain $Result(A_{empty})$ as desired.

We now prove Theorem 8 by extending the proof of Theorem 5. Again, the proof is by induction on the set of unassigned variables, i.e. $Varset - Vars$. If this set is empty, then the theorem is proved by appealing to the Lemma `check-full`. However, if `check-full` returns a conflict set, we will wish to use it to backjump instead of returning normally. Thus, we wish to pass the result of `check-full` to the continuation represented by the assumption

$$k : \forall A_0 : Assign(Vars, Valset).\, A_0 \subseteq A \to (Result(A_0) \to \bot)$$

To describe this computational behavior in the proof, we use the result of Lemma `check-full` together with k to obtain a contradiction. From this contradiction we conclude $Result(A)$ for full assignments A, so the base case is done.

In the inductive case, given a nonempty set of undefined variables *Varsleft*, we will have the inductive hypothesis

$$\begin{array}{l} IH1 : \forall s \subset Varsleft. \\ \quad \forall A' : Assign(Varset - s, Valset). \\ \quad\quad (\forall A_0 : Assign(Varset - s, Valset).\, A_0 \subseteq A' \to (Result(A_0) \to \bot)) \to \\ \quad\quad Result(A') \end{array}$$

and we must prove

$$\begin{array}{l} \forall A : Assign(Varset - Varsleft, Valset) \\ \quad (\forall A_0 : Assign(Varset - Varsleft, Valset).\, A_0 \subseteq A \to (Result(A_0) \to \bot)) \to \\ \quad\quad Result(A) \end{array}$$

This means that given $A : Assign(Varset - Varsleft, Valset)$ and the assumption

$$k : \forall A_0 : Assign(Varset - Varsleft, Valset).\, A_0 \subseteq A \to (Result(A_0) \to \bot)$$

we need to prove $Result(A)$. Here, k is a continuation function which, when given a conflict set, backjumps to the appropriate point in the search using the appropriate continuation.

As before, we will eventually construct an extension to A and use the inductive hypothesis. Note, however, that in order to apply the inductive hypothesis to an assignment A' which extends A by setting $v = n$, we must have a proof of

$$\forall A_0 : Assign((Varset - Vars) \cup \{v\}, Valset).\, A_0 \subseteq A' \to (Result(A_0) \to \bot)$$

This is the type of a new continuation function which will be built from the given function k and a continuation captured during the proof of $Result(A)$.

Again as before, we choose a variable v in *Varsleft* by applying the Lemma `choose-var`, and we use another lemma which will extend A by trying the possible values for v. The lemma in this case is

Lemma 9 *Given an assignment $A : Assign(Varset - Varsleft, Valset)$, a variable $v \in Varsleft$, and*

$$k : \forall A_0 : Assign(Varset - Varsleft, Valset).\, A_0 \subseteq A \rightarrow (Result(A_0) \rightarrow \bot)$$

then

$$\begin{array}{l} \forall Vals \subseteq Valset. \\ \quad \exists A' : Full\text{-}Assign(Varset, Valset).\, A \subseteq A' \wedge P(A') \wedge Val\_of\,(v, A') \in Vals \\ \vee \\ \quad \exists CS \subseteq Varset - Varsleft.\, \forall A' : Full\text{-}Assign(Varset, Valset). \\ \qquad Val\_of\,(v, A') \in Vals \rightarrow P(A') \rightarrow \exists v_0 \in CS.\, Val\_of\,(v_0, A') \neq Val\_of\,(v_0, A) \end{array}$$

Lemma 9 is again proved by induction on the size of the set *Vals*. Given Lemma 9, we can prove $Result(A)$ (and hence finish the proof of Theorem 8) by applying the lemma with $Vals = Valset$.

The base case of the induction is when *Vals* is empty. The proof is the same as that in the proof of Lemma 6. Since there is no full assignment extending A which gives v a value in the empty set, we must produce a conflict set. However, the property which CS must satisfy is trivial, so the empty set will suffice.

In the inductive case, we have a second inductive hypothesis:

$$\begin{array}{l} IH2(Vals) : \forall vs \subset Vals. \\ \quad \exists A' : Full\text{-}Assign(Varset, Valset).\, A \subseteq A' \wedge P(A') \wedge Val\_of\,(v, A') \in vs \\ \vee \\ \quad \exists CS \subseteq Varset - Varsleft.\, \forall A' : Full\text{-}Assign(Varset, Valset). \\ \qquad Val\_of\,(v, A') \in vs \rightarrow P(A') \rightarrow \exists v_0 \in CS.\, Val\_of\,(v_0, A') \neq Val\_of\,(v_0, A) \end{array}$$

We then wish to prove:

$$\begin{array}{l} \quad \exists A' : Full\text{-}Assign(Varset, Valset).\, A \subseteq A' \wedge P(A') \wedge Val\_of\,(v, A') \in Vals \\ \vee \\ \quad \exists CS \subseteq Varset - Varsleft.\, \forall A' : Full\text{-}Assign(Varset, Valset). \\ \qquad Val\_of\,(v, A') \in Vals \rightarrow P(A') \rightarrow \exists v_0 \in CS.\, Val\_of\,(v_0, A') \neq Val\_of\,(v_0, A) \end{array}$$

Vals represents the set of values which have yet to be tried as values of the variable v. This goal describes a partial result – either we have an assignment satisfying P or we have evidence that A cannot be extended by assigning v a value in *Vals*.

To apply the second inductive hypothesis we must reduce the set of untried values, *Vals*. So choose a value n in *Vals*, using Lemma `choose-val`, and let $A_{v=n}$ be the assignment A extended with v equal to n.

At this point in the computation, we wish to check this new partial assignment for consistency. Logically, we apply Lemma `check`. The result is either a conflict set $CS1$ which guarantees that $A_{v=n}$ cannot be extended to a satisfying assignment, or a token which signifies that search must continue.

If referring to `check` failed to produce a conflict set, we must try the partial assignments extending $A_{v=n}$. Since $A \subset A_{v=n}$, we do this by applying the first inductive hypothesis, by calling `test` recursively. We will then do a case split on the result of this call, which will be an element of $\mathit{Result}(A_{v=n})$. If a conflict set is later found which serves as evidence that $A_{v=n}$ cannot be extended, then it is to this point we expect the computation to return. So, we introduce a goal $\mathit{Result}(A_{v=n})$ which we will eventually prove by appealing to $IH1$.

First, though, in order to apply $IH1$, we must we create something of the type

$$\forall A_0 : \mathit{Assign}((\mathit{Varset} - \mathit{Vars}) \cup \{v\}, \mathit{Valset}).\, A_0 \subseteq A' \to (\mathit{Result}(A_0) \to \bot)$$

To do this, we create a continuation k' by using call/cc in the context of the goal $\mathit{Result}(A_{v=n})$ to obtain an extra assumption

$$k' : \mathit{Result}(A_{v=n}) \to \bot$$

Using k' together with the assumption k, we can create a function of the type desired. This function is intended to take an assignment and a result for that assignment and, if the result is a conflict set, backjump to the appropriate place in the search. If the result is an assignment satisfying P, then this is simply returned. There are two cases. If the result is an assignment, or if it is a conflict set which includes the variable v, then no significant backjumping occurs. Control of the program returns to the point at which v was assigned the value n, and the conflict set for the assignment A_0 (which must define a value for variable v) is returned and then modified to become a conflict set for A extended by v with respect to the set of values *Vals*. However, if the result is a conflict set not containing v, then it is in fact a conflict set for a smaller assignment, and we can use the function k to do the backjumping.

Thus, the function we create is

$$\begin{array}{l}
\lambda A_0 : \mathit{Assign}((\mathit{Varset} - \mathit{Vars}) \cup \{v\}, \mathit{Valset}). \\
\quad \lambda \mathit{result} : \mathit{Result}(A_0). \\
\qquad \mathit{case}(\mathit{result}, \\
\qquad \lambda \mathit{success} : \{S : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}) | A_0 \subseteq S \wedge P(S)\}. \\
\qquad\qquad k'(\mathit{inl}(\mathit{success})), \\
\qquad \lambda \mathit{cs} : \{CS \subseteq (\mathit{Varset} - \mathit{Vars}) \cup \{v\} | \forall A' : \mathit{Full\text{-}Assign}(\mathit{Varset}, \mathit{Valset}). \\
\qquad\qquad\quad P(A') \to \exists v_0 \in CS.\ \mathit{Val_of}\,(v_0, A') \neq \mathit{Val_of}\,(v_0, A_0)\}. \\
\qquad\qquad \mathit{if}\ (v \in \mathit{cs}) \\
\qquad\qquad \mathit{then}\ k'(\mathit{inr}(\mathit{cs})) \\
\qquad\qquad \mathit{else}\ kA(\mathit{inr}(\mathit{cs}))
\end{array}$$

This function represents only the computational content of the actual proof of

$$\forall A_0 : \mathit{Assign}((\mathit{Varset} - \mathit{Vars}) \cup \{v\}, \mathit{Valset}).\, A_0 \subseteq A' \to (\mathit{Result}(A_0) \to \bot)$$

The subset type mechanism (described in section 6) is used to hide the purely logical aspects of the proof. Thus, in introducing this function, we must prove that if v is not in cs, then $\mathit{inr}(cs)$ is a member of the type $\mathit{Result}(A)$, and other similar goals. We use *inl* and *inr* to create elements of a disjoint union and

$case(d, \lambda a.f, \lambda b.g)$ to perform a case split on an element of the disjoint union type and apply $\lambda a.f$ or $\lambda b.g$ accordingly.

Given this function, we can apply the induction hypothesis from the proof of test and produce something of type $Result(A_{v=n})$. That is, we have either an assignment A' extending $A_{v=n}$ such that $P(A')$, or it is a conflict set for $A_{v=n}$, as described above. If we have a solution, we are done. If not, then we have a conflict set (call it $CS1$) for $A_{v=n}$.

Now, given $CS1$, either from `check` or as a result of the induction hypothesis (or, computationally, from having backjumped to this point), we remove n from *Vals* and apply the second inductive hypothesis with the set $Vals - \{n\}$. If the result is an $A' : \textit{Full-Assign}(Varset, Valset)$ such that $A \subseteq A'$ and $P(A')$, then we are done. Otherwise, we have a second conflict set $CS2 \subseteq Varset - Varsleft$ satisfying

$$\begin{aligned}&\forall A' : \textit{Full-Assign}(Varset, Valset).\,(\textit{Val\_of}\,(v, A') \in Vals - \{n\}) \rightarrow \\ &\qquad P(A') \rightarrow \exists v_0 \in CS.\ \textit{Val\_of}\,(v_0, A') \neq \textit{Val\_of}\,(v_0, A)\end{aligned}$$

Now let $CS = (CS1 - \{v\}) \cup CS2$. Then $CS \subseteq Varset - Varsleft$, and it is easy to check that CS satisfies

$$\begin{aligned}&\forall A' : \textit{Full-Assign}(Varset, Valset).\,(\textit{Val\_of}\,(v, A') \in Vals) \rightarrow \\ &\qquad P(A') \rightarrow \exists v_0 \in CS.\ \textit{Val\_of}\,(v_0, A') \neq \textit{Val\_of}\,(v_0, A)\end{aligned}$$

Thus we have finished the inductive case of the lemma, and hence we have finished the proof.

6 Formal proof and program extraction

The constructive proof we have just presented proves a classically trivial theorem: every property P of variable lists either does or does not admit a satisfying assignment. The reason for presenting such a detailed proof lies in the 'proofs-as-programs' equivalence and our plan to use this equivalence to extract a correct-by-construction program for CBJ from a formal proof. Unfortunately there has been a considerable gap between the theory of proofs-as-programs and the practice of extracting usable programs from proofs. Three particular problems face us in applying our proof as a program. The first is that extracts from proofs done naively in many existing automated proof assistants are not usable in practice because they contain huge amounts of non-computational material. This problem is endemic to constructive proof systems. Next, in any case, the computational facilities of these systems do not support reasoning or computing with the classical control operator call/cc. Finally, while a proof in a formal system ensures correctness of the extracted program (*i.e.* the extension of the program is correct) , it is not clear with current methods how to guarantee that the extracted program implements the intended algorithm. This is the problem of intensional analysis and the methods for its solutions are less clear.

One approach to solving these problems is to base application code on an informal proof such as the one presented above in Section 5. This is the approach taken by Gent and Underwood in [21]. However, the process of translation leaves considerable room for errors. Nevertheless, profiling execution of concrete code provides a means to determine whether the algorithm behaves as expected.

In this paper we tighten the link between the proof and the code by formalizing the proof in the Nuprl system [14]. We take advantage of recent advances in the extraction of programs from Nuprl proofs [8, 9] to extract a program that is virtually indistinguishable from a functional program written in an ordinary programming language such as Scheme or Lisp. To take advantage of the efficiencies offered by modern compiler technology, we translate the extracted program into Scheme in a natural way. The Appendix includes a discussion of the translation and the texts of both programs.

Nuprl is an implementation of an extended Martin-Löf type theory [32]. The Nuprl type system is extremely rich. For the purposes of this paper we use it with little explanation but hope interested readers will seek further information elsewhere. The Nuprl book [14] offers a detailed description of the type theory. Also see [28]. As in other constructive type theories, in Nuprl we can extract a program (i.e. something which is meant to be interpreted computationally) from proofs in the logic. Three aspects of Nuprl are of particular interest for the purposes of this paper.

First, Nuprl includes a *set type*, that is, types of the form $\{$`y:T | P[y]`$\}$ where `T` is a type and `P[y]` is a proposition. The members of the type are elements `a` of type `T` such that `P[a]` holds. It should be remarked that although the notation suggests ordinary set comprehension, in Nuprl it denotes a type, not a set. Set types are closely related to the constructive existential type (or sigma type) but they are distinguished from them by the form of their inhabitants. Inhabitants of $\exists$`y:T.P[y]` are pairs $\langle$`a,t`$\rangle$ where `a` is an element of type `T` such that `P[a]` holds, and `t` is a term witnessing the proof of `P[a]`. Often, the second component of an existential witness has no computational interest. Indeed, these terms are often extremely large, obfuscating the structure of the extracted program. Furthermore, they often add significantly to the computational complexity of the extracted program. However, it should be noted that set types do not come for free; computationally, given an element `x` of $\{$`y:T | P[y]`$\}$, we may use `x` freely but not the proof of `P[x]`. This constraint leads to technical complications on the proof side which are addressed by methods described in [8, 9, 10].

The second aspect of Nuprl that is of interest to our efforts is that its computation system is untyped. The typing rules describe when a term inhabits a type, and are expressive enough to permit typing judgements about terms without always having to assign types to every subterm. In particular, general recursive functions can be defined using Curry's Y combinator [27]. This method is useful for generating readable and efficient extracts. Additionally, theorems stating induction principles can be proved so that the computational content of their proofs are efficient recursion schemes. The typing ensures that the recursion is well-founded. In [8] a `letrec` form is introduced in terms of the Y combinator, so that extracts need not display the recursion mechanism explicitly.

The third aspect of Nuprl that makes it amenable to our purposes is our ability to extend the system to allow for reasoning about call/cc. This was accomplished by making a classical extension to the system. We added a rule corresponding to double negation elimination and specified the extract of invocations of the rule to be the term call/cc. The extension is semantically justified by [33, 34]. As an example application of the rule we show the a form of Peirce's law and its extract.

```
* THM Peirces_law
∀T:𝕌. ((T ⇒ False) ⇒ T) ⇒ T
Extraction:
  λT,%.call/cc(λk.% k)
```

We did not extend the Nuprl evaluator to include evaluation rules for call/cc, this is future work.

These properties of Nuprl make it possible to extract terms from proofs that are very nearly ordinary functional programs.

The Nuprl formalization differs from the informal proof presented above in three ways. In order to exploit existing Nuprl libraries, we specify the problem using lists as our concrete representation of finite sets. We also restrict ourselves to two-valued assignments eliminating the second induction corresponding to (Lemma `enumerate-domain`) and `choose-val`; instead, we enumerate the values explicitly in the Nuprl proof. Finally, we have not included the check on partial assignments.

This final restriction may seem to be a significant drawback because partial assignments will often fail checks when many variables are unassigned. However, when using backjumping, the absence of `check` need not increase the amount of search. In the implementation of a solution to the directed Hamiltonian Circuit problem described below, `check-full` is implemented to check assignments in the order they were constructed. Thus, the conflict set reported on a branch of the search tree is always the one that would have been reported by the first failed `check`. Backjumping immediately jumps from the bottom of the branch to this point and resets the value. Therefore, the number of branches searched is the same as it would be with `check` in place. An inefficiency that does result from this restricted version is that `check-full` duplicates computations across branches, for example the empty partial assignment is checked on every branch of the search tree.

In any case, the proof formalized in Nuprl includes as an instance the implementation of a solution to the Directed Hamiltonian Circuit problem described below and encompasses a large class of search problems.

The Nuprl formalization ensures that we have precise logical characterizations of the problem specific functions corresponding to the lemmas `choose-var` and `check-full`. An instance of the extracted program is guaranteed to be a correct search procedure as long as these functions meet their specifications.

The computational content of `choose-var` (as applied in the Theorem 8) is formalized here as a function which decomposes nonempty lists (our concrete representation of sets.) We defined its type as follows.

```
CHOOSE(T) == {f:T List+ →(T List × T × T List)|is_decomp(T;f)}
```

where

```
is_decomp(T;f) ==  ∀L:T List+. let M,u,N = (f L) in L = M @ (u::N)
```

Thus, CHOOSE(T) is the type of functions from non-empty T lists L to triples ⟨M,u,N⟩ such that L = append(M ,cons(u,N)). The Nuprl theorems are stated so that a function of this type is a parameter to the extracted program.

We also develop the following induction machinery. The induction used in the proof is on the list of variables which have not yet been assigned values. The following theorem is a general induction principle on T lists that is parameterized by a function of CHOOSE(T). This allows for list decomposition based on a user defined selection criteria which may depend on properties of the entire list.

```
* THM sublist_ind
∀T:𝕌.
 ∀choose: CHOOSE(T) .
  ∀P:T List → ℙ.
    P[[]] ⇒
     (∀L:T List+. let M,u,N = (choose(L)) in P[M @ N] ⇒ P[L]) ⇒
   (∀L:T List. P[L])
```

The proof of the theorem yields the following extract term which is a recursion scheme corresponding to list induction.

```
  λT,choose,P,b,g.
   (letrec f(L) =
     if null(L) then b
     else let M,u,N = choose(L) in g(L)(f(M @ N))
     fi)
```

To read the extract notice that b corresponds to the base case, (*i.e.* the computational content of the assumption P[[]]) and g corresponds to the computational content of the induction hypothesis. Together with this new induction principle we have defined a tactic which automates the use of this theorem in proofs.

In the Nuprl proof, Assignments are defined to be functions from the type of variables Var to a three element type $\mathbb{N}_3$ = $\{0_3, 1_3, 2_3\}$. These three values are interpreted as "false", "undefined", and "true" respectively.

Recall from the discussion of predicates and conflict sets in Section 3.1 that the predicate P is assumed to be decidable on full assignments, *i.e.* on *Full-Assign*(*Varset, Valset*). Since we will represent the set of variables *Varset* as a list we define full assignments as follows.

```
Full[L] ==  {a:assignment| ∀x∈L. defined(a x)}
```

Using this type we characterize predicates as functions of the following type,

```
(Varset:Var List → Full[Varset] → 𝔹)
```

The fact that functions of this type are Boolean valued means they are decidable and the restriction to assignments of type `Full[Varset]` guarantees that they are defined on full assignments, but not on partial assignments.

We now define a type which corresponds to the lemma `check-full` as used above.

```
 CHECKFULL Type ==
 P:(Varset:Var List → Full[Varset] → 𝔹) →
   Varset:Var List → a:Full[Varset] → Result(P;Varset;a)
```

Thus `CHECKFULL Type` is the type of functions accepting a decidable predicate `P`, a list of variables `Varset`, a full assignment `a` and returns something of type `Result(P;Varset;a)`. `Result` corresponds to the abbreviation *Result*(a) defined in the course of the proof of Theorem 5.

Its formalization in Nuprl appears as follows.

```
Result(P;L;a) ==  ASet(P;L;a) ∨ CSet(P;L;a)
```

where

```
ASet(P;L;a) == {a':Full[L]| a⊆a' ∧ P[L;a']},
CSet(P;L;a) == {cs:MSet[L;a]|
                 ∀a':Full[L]. P[L;a'] ⇒ ∃v∈cs.(¬(a v = a' v))},
 and
MSet[L;a] == {M:Var List| M(⊆=v)L c∧ ∀x∈M. defined(a x)}
```

Note the use of set types in place of existentials, this eliminates logical content from the extract. Thus, elements of `Result(P;L;a)` is either a term of the form `inl(a')` or `inr(cs)` where `a'` is a full assignment inhabiting `ASet(P;L;a)` and `cs` is a conflict set inhabiting `CSet(P;L;a)`.

Given these definitions we have enough to state the Nuprl theorem roughly corresponding to Theorem 8.

```
* THM find
∀choose: CHOOSE(Var).
 ∀checkfull: CHECKFULL Type.
  ∀P:L:Var List → Full[L] → 𝔹.
   ∀L:Var List.
    ∀a:assignment.
    (∀a0:{a0:assignment| a0⊆a} . Result(P;L;a0) ⇒ False) ⇒
    Result(P;L;a)
```

The formal proof of this theorem follows closely the proof of Theorem 8 presented above. We do not present it here, but instead turn the reader's attention to the Appendix which contains the program extracted from the proof.

Although an extract from a complete proof is guaranteed to be a correct search algorithm, it is more difficult to determine if it is in fact the desired search algorithm. This is particularly true if the desired algorithm is described informally or in a different setting. To check the theoretical development, we used the extract as a basis for a concrete implementation. This work is described in the next section.

7 Implementation Example: Hamiltonian Circuit

We have constructed a proof that corresponds to conflict-directed backjumping, and formalized this proof in Nuprl to yield an almost entirely computational extract. To show the usefulness of this extract, we used it to implement a novel algorithm, namely conflict-directed backjumping (CBJ) for the Hamiltonian Circuit problem. We do not know of any previous use of CBJ or any other intelligent backtracking technique having been reported in this domain.

We chose to use the language Scheme [12]. We include the code corresponding to the proof extract in an appendix, and brief notes on the correspondence between the code and the Nuprl extract. There were two reasons for not using the execution facilities in Nuprl. First, Nuprl does not have any computational equivalent of the control operator call/cc, whereas it is available in Scheme. Second, while we do not see fundamental difficulties in doing so, we have not attempted to discharge the proof obligations imposed by our proof on the various functions we have written for the Hamiltonian circuit problem. Instead, we view this implementation example as a demonstration of using our general-purpose Scheme code to develop a rapid prototype of CBJ in new problem classes. The central idea of conflict-directed backjumping is sufficiently subtle that it is not easy to see how to apply it to new domains. We have eliminated this difficulty.

We consider the directed Hamiltonian circuit problem, to find a permutation of nodes in a directed graph such that there is an edge between each consecutive pair of nodes in the permutation, and between the last and first nodes in the permutation. A natural formulation is for Boolean variables to correspond to edges in the graph. A true variable corresponds to an edge chosen to be in the circuit, while a false variable corresponds to an edge not in the circuit. We did not implement any intelligent variable selection heuristic, but merely pick edges in lexicographic order – we numbered nodes arbitrarily and pick an edge from the lowest remaining node in this order. As a value ordering heuristic we always choose to set each variable false before true.

It remains to implement the functions that check assignments and to return appropriate conflict sets. Fortunately, the Hamiltonian circuit problem can be captured by three simple rules:

- Each node must have at least one edge coming into it. If this condition is violated then all variables representing edges coming into the node must have been set to false. The value of at least one such variable must be reset to true, so this set of variables is a valid conflict set. A similar rule applies to edges leaving a node.

- Each node must have no more than one edge coming into it. If an assignment breaks this condition, some pair of variables representing edges into a given node must both be set true. The value of at least one of these two variables must be reset to false, so the pair of variables is a valid conflict set. The equivalent rule applies to edges leaving the node.
- The previous two conditions ensure that the edges chosen in a full assignment must form a number of circuits comprising all edges. However it does not ensure that there is only one global circuit: there may be a number of sub-circuits. So the final rule is that no set of variables representing a circuit of nodes may all be true, unless the circuit involves all nodes in the graph. If this condition is violated we must reset one of the values to false, so the set of variables in the sub-circuit is a valid conflict set.

Given these rules, it was straightforward to implement Scheme functions that checked them given a particular graph and partial assignment, and either indicated that the check was passed, or indicated failure and returned a conflict set.

We tested our implementation on small Hamiltonian circuit problems on directed graphs with 10 nodes and 36 edges. We generated 100 such graphs randomly. This was done simply by picking an edge at random from the 90 possible directed edges, then a second edge at random from the 89 remaining, and so on. Of our graphs, 78 had circuits while 22 did not. Such data sets from a 'phase transition region', with a mixture of soluble and insoluble problems, are often used for benchmarking algorithms [11].

It is interesting to compare the number of branches searched with and without backjumping. Assuming Kondrak and van Beek's results for binary constraint satisfaction problems [29] extend to the general case of CBJ, the use of backjumping while using the same heuristics and checking mechanism should never increase the number of branches searched, while possibly decreasing it, compared to simple backtracking. To test this, we implemented the same heuristic and checking functions for a backtracking algorithm. The mean number of branches searched by backtracking was 123.81, with a worst case of 824 branches. In contrast, the mean number of branches searched by CBJ was 38.18, with a worst case of 197 branches. As expected, in no case did CBJ search more branches than backtracking. Although there were a number of cases where no reduction in search was achieved, these were all on problems solved quickly by both algorithms: in every case where backtracking needed more than 20 branches, CBJ was able to search fewer branches. In some cases the reduction was particularly dramatic: for example one insoluble problem required 614 branches with backtracking but only 10 with conflict-directed backjumping. Our results suggest that conflict-directed backjumping is a worthwhile technique for the Hamiltonian circuit problem.

We tested the same graphs with an implementation of Martello's algorithm [31]: on each of the 100 problems this reported the same status as our implementation, suggesting correctness of our implementation of the functions for Hamiltonian circuit. Martello's algorithm includes two features not present in our implementation of CBJ: an effective dynamic variable ordering heuristic, and propagation techniques. The heuristic is to pick constrained edges first, for example edges going to nodes with small in-degree: this is normally more effective than an an arbitrary lexicographic order. The propagation techniques can, for example, force an edge to be in the circuit if it the only edge coming out of a certain node. The result is that on our problems Martello's algorithm requires a mean of only 2.82 branches with a worst case of only 7 branches. Incorporating CBJ into algorithms such as Martello's should bring additional reductions in the amount of search. While we have not yet developed code formally to test this conjecture, experiments on hand-written code strongly suggest it is true.

7.1 Implementation Efficiency

There is no reason why code developed formally need be slower than any other code. Indeed, call/cc in Scheme is so suitable for implementing backtracking algorithms that code based on it is likely to be faster than Scheme code implemented differently. In a previous study, code developed formally was marginally faster than the original implementation of CBJ in Scheme for constraint satisfaction problems [21]. In the case of Hamiltonian Circuit, no such comparison is possible since CBJ has not previously been implemented for it.

To give some idea of relative efficiency, we compared run times of our code with a Common Lisp implementation of Martello's algorithm. To give the closest comparison we translated our Scheme code into Common Lisp. As dialects of Lisp, the translation between these languages is mostly straightforward, except that call/cc is not available in Common Lisp. Fortunately, Norvig has described a simple implementation of call/cc in Common Lisp when (as here) the created continuation only has dynamic extent [35]. Our translated code produced identical results to that of our Scheme code.

On our 100 graphs Martello's algorithm required a mean of 0.0850 cpu seconds per instance on a DEC Alpha 3000-300LX 125MHz running Gnu Common Lisp. Our implementation of CBJ took a mean of 3.24 seconds on the same machine. Thus our code took just under 40 times as long. This factor should be balanced by the fact that CBJ had to search about 13 times as many branches than Martello's algorithm. It would be naive to suggest that our code is about 3 times slower per branch, but the comparison does show that our code is not ridiculously slow on the small problems we tested.

A fairly small run time overhead is reassuring, but we believe that with further work we could eliminate it completely. As well as a general lack of attention to implementation efficiency in the supporting functions for Hamiltonian circuit, there are two specific areas in which our code could be improved. First, as described earlier, the absence of `check` in the formal Nuprl proof meant that nodes high in the search tree can be re-checked many times. This would be alleviated by extending the Nuprl formalization to incorporate this optimization. Second, our implemented code for the Hamiltonian circuit does not cache work done to check one partial assignment, in order to check later assignments faster. However, in many domains subtle use of data structures is what allows fast code to be implemented for search algorithms. Again, there is nothing intrinsic in our methodology which forces this inefficiency. In particular, we have been very free in our definition of the assignment type in section 3.1. The only properties we have assumed about assignments are that we can order them (by prefix or subset) so we can say $A' \subseteq A$, and that we can look up the value of a variable v in an assignment A using $\mathit{Val\_of}\,(v, A)$. The actual type of an assignment may be much more complicated – it may, for instance, contain caches of information about expensive past computations in order to save recomputing them. Such information could be computed by the `check` and `check-full` functions, and returned to the main function by returning a (possibly modified) assignment structure to the one passed to it. All that is necessary is that the two structures satisfy the observational equality that all values of $\mathit{Val\_of}\,(v, A)$ are identical. Beyond that the implementation of the checking functions would be free to change the structure of the assignment. This idea can be extended to allow a method used in many of the most efficient implementations of search algorithms. This is to change internal data structures when moving in both directions in the search tree, for example changing values in an array when a variable is set, then later undoing this change on backtracking. In our framework this could equally be done by a slight change to the continuation created when backjumping is necessary, and an additional obligation on the implementer to create a function to undo variable assignments: each time this takes one step further back the search tree, the undo function would be called.

To summarize, we have shown that our general methodology has allowed the implementation of conflict-directed backjumping for the Hamiltonian circuit problem, even though we believe that CBJ has never previously been described in this domain. Our implementation achieved significant reductions in search compared to a backtracking algorithm. While our code did run slower than a previously described algorithm, there is nothing essential to our methodology which makes this necessary.

8 Related and future work

Following Prosser's introduction of conflict-directed backjumping (CBJ) [40], Ginsberg [22] and Kondrak and van Beek [29] have given proofs of the correctness of CBJ and also related the numbers of nodes searched by different algorithms. The significant advance of our work is in its underlying basis in formal semantics and in its generality. Ginsberg gave proofs of pseudo-code written in English, and Kondrak and van Beek of Prosser's Pascal-like pseudo-code: thus neither proof applies to code for which formal semantics exists. Our results are very general because they apply to a wide variety of search algorithms, and a wide variety of problem classes, all obtainable from the Scheme code we have presented by implementing suitable service functions.

Related work on formal development of search algorithms by Smith et. al. [7, 37, 44] has concentrated on techniques for transforming search problem specifications into executable search procedures. These techniques make use of a deep analysis of the structure of the problem specification to produce very efficient code tuned to the particular constraints involved. In contrast, our method is independent of the details of the problem class as long as a solution can be determined by a predicate on assignments. Thus our approaches are complementary; it is possible that the problem analysis techniques could generate very efficient functions for testing possible solutions (`check`, `check-full`) or variable- and value- ordering heuristics (`choose-var`, `choose-val`).

One of the interesting questions our work raises is how to distinguish between correct algorithms for the same problem. Our work has focussed on the algorithm CBJ, but our proof only formally shows that we have a correct algorithm for solving search problems. Other proofs would correspond to other algorithms, for example simple enumeration or naive backtracking. Of course any two correct algorithms must by definition have identical input/output behaviors, but one may need much more search to solve the same problems. Choosing an appropriate search algorithm is often the key step in solving combinatorial problems. Kondrak and van Beek have classified a variety of constraint satisfaction algorithms and related the numbers of nodes searched by different algorithms [29]. It would be interesting to generalize this work within our framework, relating algorithms formally proven correct and very generally applicable. This would go some way to the problem of distinguishing between algorithms.

However, the very generality of our approach means that what might be seen as very different algorithms can be implemented by the provision of different checking functions to the single extract we have proved in this paper. The proof obligations we have specified in Lemmas 2, 3, 4 and 7 are sufficient, if fulfilled, to guarantee correctness of search for a satisfying assignment. The proof we have given and the resulting extracted λ-terms naturally implement CBJ. However, our proof obligations are designed to ensure correctness of the resulting algorithm, rather than guaranteeing that a particular intended algorithm has in fact been implemented. Depending on how the obligations are fulfilled, our code, while still correct, may search in the manner of algorithms different from CBJ. This is not a serious concern, as the most natural implementation satisfying the proof obligations together with the extract given in this paper, will typically result in CBJ, as intended. However we now discuss some of the issues our observations raise, and how these can be addressed in future work.

The principal freedom that we give in fulfilling the proof obligations arise in `check`. These obligations can be filled either arbitrarily weakly or arbitrarily strongly.

An implementation of `check` which always returns the unit token indicating continued search completely satisfies the proof obligations. Given a correct implementation of `check-full`, sound and complete search is carried out, but by enumerating all full assignments. Even given a less naive implementation, if the proof obligations are fulfilled by returning one conflict set, any superset of that conflict set and subset of *Vars* trivially meets the proof obligations. In particular, if a partial assignment $A : Assign(Vars, Valset)$ cannot be extended to a satisfying assignment, the set *Vars* itself is always a valid conflict set. But returning all variables in *Vars* as a conflict set disables all backjumping beyond trivial backtracking. The result is again a correct search algorithm, but backtracking rather than conflict-directed backjumping.

As well as implementations which satisfy proof obligations weakly, proof obligations can be satisfied very strongly. For example, any decision procedure may be implemented for the problem at hand. This can be applied at the root of search, i.e. with the empty partial assignment. If the decision procedure shows that the problem has no solution, it may correctly return an empty conflict set. Otherwise 'search' continues. Naturally this use of an oracle is not the intended application of our work, but is entirely legal. Less trivially, at any point some degree of work can be carried out to determine if the current partial assignment can extend to a solution. Doing this, search may terminate earlier than in a more straightforward implementation. Indeed, some other sophisticated search algorithms do exactly this: examples are Forward Checking (FC) [25] and Maintaining Arc Consistency (MAC) [42] both of which can be merged with CBJ [40, 23]. These techniques could be implemented directly in our framework. In particular, just as assignments can be generalized to cache computations (as described in section 7), so information about impossible values and conflict sets for future variables could be stored in generalized assignments. Indeed, in an earlier paper the second and third authors reported on the implementation of the equivalent of FC+CBJ for propositional satisfiability [21] in a framework similar to that of this paper. However, our framework is not the best for implementation of lookahead techniques in general. It would be better to capture the general nature of lookahead search, just as we have done for backjumping search in this paper. Applying the methodology used in this paper to the proof of algorithms which naturally combines lookahead and backjumping search remains interesting future work.

Another area yet to be explored fully is the application of these ideas to backtracking proof search procedures like tableaux. When tableau search is constructed as a search in parallel for a proof and a counter-model [46], it has the same logical structure as the search described in section 2. It may be possible to use conflict-directed backjumping in conjunction with information obtained from one branch of the tableau to eliminate search in other branches of the tableau and to reduce the size of the proof constructed as a result of the search.

Finally, the Nuprl proof could be generalized to more abstract types for sets and assignments. In earlier work [21], two of the authors formalized the core of this work in Lego [30, 38], an implementation of type theory based on an extension of the calculus of constructions [15]. Lego does not have Nuprl's sophisticated program extraction mechanisms, so the result was not so closely connected to Scheme code. However, the approach taken included a very abstract approach to the underlying data types, which essentially entailed a specification of abstract data types for sets and assignments. This work should be easily transferable to the setting of Nuprl.

9 Conclusion

In this paper, we have presented a very general view of search applicable in many contexts, including proof search in decidable theories. The search is based on extending a partial structure finitely often until either it satisfies a specified predicate or there is evidence that it fails to satisfy the predicate. We have shown how the evidence of failure can be used to reduce the need to search other extensions of the partial structure.

We have demonstrated this in detail in the case of search for assignments of values to variables satisfying some predicate: all NP complete problems are instances of this and thus such search problems are of enormous practical import. Using the proofs-as-programs paradigm, we have shown that careful reasoning about 'conflict sets' can be used to derive a proof corresponding to the search algorithm 'conflict-directed backjumping' (CBJ). Our proof uses the classical typing of the nonlocal control operator call/cc, demonstrating the practicality of classical typing for describing typical uses of nonlocal control. The formalization of this proof in Nuprl further advances the technology for creating recognizable programs from proofs with computational content.

We have developed Scheme code based on our proof, and used it to show the value of CBJ for the Hamiltonian circuit problem. Our work shows that sophisticated search techniques can be proved correct very rigorously and at a high level of abstraction, yet sufficiently concretely to allow their immediate application to domains in which they have not previously been used.

References

[1] A.B. Baker. Intelligent backtracking on constraint satisfaction problems: Experimental and theoretical results. Technical report CIS-TR-95-08, Computer and Information Sciences, University of Oregon, 1995.

[2] F. Barbanera and S. Berardi. Witness extraction in classical logic through normalization. In G. Huet and G. Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[3] F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Information and Computation*, 125:103–117, 1996.

[4] R.J. Bayardo Jr. and R.C. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In E.C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, pages 46–60. Springer, Berlin, 1996.

[5] R.J. Bayardo Jr. and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208, Menlo Park, CA, 1997. AAAI Press.

[6] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity*, pages 77–97, Berlin, 1994. Springer.

[7] M. B. Burstein and D. R. Smith. Itas: A portable interactive transportation scheduling tool using a search engine generated from formal specifications. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, 1996.

[8] J. L. Caldwell. Moving proofs-as-programs into practice. In *12th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 1997.

[9] J.L. Caldwell. Classical propositional decidability via Nuprl proof extraction. In *TPHOLs'98 The 11th International Conference on Theorem Proving in Higher Order Logics*. Springer Verlag, 1998. To Appear.

[10] J.L. Caldwell. *Decidability Extracted: extracting tableau procedures for classical and intuitionistic propositional logic from formal proofs*. PhD thesis, Cornell University, 1998. To appear.

[11] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the Twelfth International Conference on Artificial Intelligence*, pages 331–337, San Mateo, CA, 1991. Morgan Kauffman.

[12] W. Clinger and J.A. Rees. The revised^4 report on the algorithmic language scheme. *ACM LISP Pointers*, 4(3), 1991.

[13] R. L. Constable and D. J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.

[14] R.L. Constable, S.F. Allen, W.R. Cleveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with The Nuprl Development System*. Prentice-Hall, New Jersey, 1986.

[15] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[16] O. Danvy and A. Filinski. Abstracting control. In *Lisp and Functional Programming*, pages 151–160. ACM Press, 1990.

[17] R.K. Dybvig and R. Hieb. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990. Also Indiana University Computer Science Department Technical Report #256.

[18] D.P. Friedman, C.T. Haynes, and E.E. Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 263–274. Springer-Verlag, 1984.

[19] M. R. Garey and D. S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness.* W H Freeman, New York, 1979.

[20] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[21] I.P. Gent and J. Underwood. The logic of search algorithms: Theory and applications. In G. Smolka, editor, *Principles and Practice of Constraint Programming – CP97*, pages 77–91, Berlin, 1997. Springer.

[22] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[23] S.A. Grant and B.M. Smith. The phase transition behaviour of maintaining arc consistency. In W. Wahlster, editor, *12th European Conference on Artificial Intelligence*, pages 175–179, Chichester, 1996. John Wiley & Sons.

[24] T. Griffin. A formulas-as-types notion of control. In *Proc. of the Seventeeth Annual Symp. on Principles of Programming Languages*, pages 47–58, New York, 1990. Association for Computing Machinery.

[25] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[26] R. Harper, B.F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3:465–484, 1993.

[27] D.J. Howe. Reasoning about functional programs in Nuprl. In P.F. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 145–164. Springer-Verlag, 1993.

[28] Paul Jackson. The Nuprl proof developemnt system, version 4.2 reference manual and user's guide. Computer Science Department, Cornell University, Ithaca, N.Y. Manuscript available at `http://www.cs.cornell.edu/Info/Projects/NuPrl/`, July 1995.

[29] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[30] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[31] S. Martello. An enumerative algorithm for finding Hamiltonian circuits in a directed graph. *ACM Transactions on Mathematical Software*, 9:131–138, 1983.

[32] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.

[33] C.R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, 1990. Also Department of Computer Science technical report TR90-1151.

[34] C.R. Murthy. An evaluation semantics for classical proofs. In *Proceedings 6th Annual IEEE Symp. on Logic in Computer Science*, pages 96–107. IEEE, Los Alamitos, CA, 1991.

[35] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kauffman, San Mateo, CA, 1992.

[36] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings International Conference on Logic Programming and Automated Reasoning, LPAR'92*, pages 190–201. Springer, Berlin, 1992.

[37] P. Pepper and D. R. Smith. A high-level derivation of global search algorithms (with constraint propagation. *Science of Computer Programming, Special Issue on FMTA (Formal Methods: Theory and Applications*, 1996.

[38] R. Pollack. *The Theory of Lego*. PhD thesis, University of Edinburgh, 1995. Available as report ECS-LFCS-95-323.

[39] P. Prosser. *Distributed Asynchronous Scheduling*. PhD thesis, Strathclyde University, 1990.

[40] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[41] J.-F. Puget. Applications of constraint programming. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming – CP95*, pages 647–650, Berlin, 1995. Springer.

[42] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, Chichester, 1994.

[43] B.M. Smith and S.A. Grant. Sparse constraint graphs and exceptionally hard problems. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 646–651, San Mateo, CA, 1995. Morgan Kauffman.

[44] D. R. Smith, E. A. Parra, and S. J. Westfold. Synthesis of planning and scheduling software. In A. Tate, editor, *Advanced Planning Technology*. AAAI Press, 1996.

[45] R.M. Smullyan. *First Order Logic.* Springer-Verlag, Berlin, 1968.

[46] J.L. Underwood. The tableau algorithm for intuitionistic propositional calculus as a constructive completeness proof. In *Proceedings of the Workshop on Theorem Proving with Analytic Tableaux, Marseille, France*, pages 245–248, 1993. Available as Technical Report MPI-I-93-213 Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[47] J.L. Underwood. *Aspects of the Computational Content of Proofs.* PhD thesis, Cornell University, 1994. Also Department of Computer Science technical report TR94-1460.

Appendix: Nuprl extract and Scheme code

In this appendix we present the Nuprl extract described in section 6 and the Scheme code used in the implementation reported in section 7.

Nuprl's evaluation semantics is lazy while Scheme's evaluator is eager. Thus, the naive translation of Nuprl extracts into Scheme programs used here may not, in general, result in a correct Scheme program. It is possible for a term (say p) extracted from a Nuprl proof to contain non-terminating sub-terms. Even if lazy evaluation of p always leads to termination, the naive translation of p may not. It is also possible for p to contain redices which are not type safe, but which will never be reduced under the lazy semantics. The naive translation of such a program into Scheme may result in Scheme crashing.

Examination of the Nuprl extract presented below reveals that it does not have either of these bad properties. Thus, the naive translation, essentially an operator for operator translation of the Nuprl code into Scheme is faithful to the Nuprl original. In the main, the only differences are syntactic, but a few others may be less obvious. There is no equivalent in Scheme of disjoint union types; to work round this we give our own definitions of `inl`, `inr` and `case`. Similarly multiple values cannot be returned directly in Scheme, hence we wrote `choose` so that it returned a three element dotted list of all elements up to the chosen one, the chosen element itself, and all elements after it. These elements can then be accessed by `car`, `cadr` or `cddr`. In the Nuprl extract, $\mathbb{N}_3$ is a type containing three elements, the element 0_3 is translated to `false`, 2_3 is translated to `true`, and 1_3, used for the unassigned value, is represented in Scheme by the atom `'unassigned`. The use of `letrec` is different in the two formalisms, leading to an extra explicit call to `test` in the Scheme code: this happens implicitly in the Nuprl.

The Nuprl code displayed here has been automatically cleaned up by one application of the rewrite conversion `Reduce` and after folding one operator defintion (`spread3`).

The Nuprl code includes two artifacts explained by the fact that it is extracted from a proof. The first is that the function bodies under the top level `then` and `else` clauses accept three arguments. The first two are the assignment and the continuation respectively, but the third (named `%` and `%1` by the system) correspond to a logical part of the proof. Fortunately, their arguments in this extract turn out to be the trivial term `Ax`. These extra parameters and their arguments were not preserved in the Scheme translation. The second artifact of the extraction is that the first argument in the recursive calls to test is the term `(choose(L0).1 @ choose(L0).3)` (the append of the first and third elements of `choose(L0)`) instead of the equivalent, and more natural `(y1 @ y4)`. This foible of extraction is preserved in the Scheme translation only to aid in the comparison.

As an artifact of using lists to represent sets, we must ensure that the list of unassigned variables has no repeated elements, this arises in the extract as an application of the list function `unique`, which deletes duplicate elements of a list. The Scheme translation correctly preserves this.

```
λchoose,checkfull,P,L,a,kk.
 (letrec test(L0) =
   if null(L0)
   then λa,kk,%. kk(a)(checkfull(P)(L)(a))
   else λa,kk,%1.
          let y1,y3,y4 = choose(L0) in
             case call/cc (λ%7.
                   test((choose(L0)).1 @ (choose(L0)).3)
                       (λx.if =v(x)(y3) then 0₃ else a(x) fi )
                       (λa0,r_a0.
                          case r_a0
                          of inl(aa) => %7(inl(aa))
                           | inr(cs) => if y3∈cs
                                        then %7(inr(remove(=v;y3;cs)))
                                        else kk(a)(r_a0)
                                        fi )
                       (Ax))
            of inl(%8) => inl %8
             | inr(%9) => case call/cc(λ%11.
                                 test((choose(L0)).1 @ ((choose(L0))).3)
                                     (λx.if =v(x)(y3) then 2₃ else a x fi )
                                     (λa0,r_a0.
                                        case r_a0
                                        of inl(aa) => %11(in(aa))
                                         | inr(cs) =>
                                            if y3∈cs
                                            then %11(inr(remove(=v ;y3;cs)))
                                            else kk(a)(r_a0)
                                            fi)
                                     (Ax))
                        of inl(%12) => inl(%12)
                         | inr(%13) => inr(kk(a)(inr(remove(=v ;y3;(%9 @ %13)))))
   fi
 )(unique(filter(λx.undefined(a x);L)))(a)(kk)(Ax)
```

```
(define case (lambda (elt case1 case2)
      (if (car elt) (case1 (cdr elt)) (case2 (cdr elt)))))

(define inl (lambda (arg) (cons #t arg)))

(define inr (lambda (arg) (cons #f arg)))

(define find_ext
(lambda (choose checkfull P L a kk)
  (letrec ((test (lambda (L0) (if (null? L0)
  (lambda (a kk) (kk a (checkfull P L a)))
  (lambda (a kk)
    (let* ((y1y3y4 (choose L0)) (y3 (cadr y1y3y4)))
         (case (call/cc
                (lambda (%7) ((test (append (car (choose L0)) (cddr (choose L0))))
                  (lambda (x) (if (equal? x y3) #f (a x)))
                  (lambda (a0 r_a0)
                    (case r_a0
                    (lambda (aa) (%7 (inl aa)))
                    (lambda (cs) (if (member y3 cs)
                                    (%7 (inr (remove y3 cs)))
                                    (kk a r_a0)
                                ))))
                  )))
         (lambda (%8) (inl %8))
         (lambda (%9) (case (call/cc
                             (lambda (%11) ((test (append (car (choose L0)) (cddr (choose L0))))
                               (lambda (x) (if (equal? x y3) #t (a x)))
                               (lambda (a0 r_a0)
                                 (case r_a0
                                 (lambda (aa) (%11 (inl aa)))
                                 (lambda (cs) (if (member y3 cs)
                                                 (%11 (inr (remove y3 cs)))
                                                 (kk a r_a0)
                                             ))))
                               )))
                    (lambda (%12) (inl %12))
                    (lambda (%13) (inr (kk a (inr (remove y3 (append %9 %13))))))
)))))))))
((test (unique (filter (lambda (x) (equal? (a x) 'unassigned)) L)))
a
kk
))))
```