Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 3–38
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Programmed Strategies for Program Verification

Richard B. Kieburtz

Portland State University
and
OGI School of Science & Engineering, OHSU
Portland, Oregon, USA

Abstract

Plover is an automated property-verifier for Haskell programs that has been under development for the past three years as a component of the Programatica project. In Programatica, predicate definitions and property assertions written in *P-logic*, a programming logic for Haskell, can be embedded in the text of a Haskell program module. Properties refine the type system of Haskell but cannot be verified by type-checking alone; a more powerful logical verifier is needed.
Plover codes the proof rules of P-logic, and additionally, embeds strategies and decision procedures for their application and discharge. It integrates a reduction system that implements a rewriting semantics for Haskell terms with a congruence-closure algorithm that supports reasoning with equality. It employs strategies such as structure splitting and case analysis to explore alternative valuations of expressions of type Bool or other finite data types, but these strategies can lead to exponential growth of terms and must be employed cautiously.
Plover itself is written in Stratego, which has proven to be a powerful language tool for implementating a verifier. We discuss the design and implementation of some strategies that enable Plover to comprehend Haskell and verify many valid property assertions.

Keywords: Haskell, verification, logic, rewriting, strategies, Stratego, decision procedures, reduction, normal forms, theorem proving

1 Introduction

This paper describes the architecture and implementation of an automatic property verification tool for Haskell programs. This tool, called *Plover*, is being developed as part of the Programatica project, whose objective is to explore means of providing scientifically based certification of formally specified properties of computer programs.

A Programatica certificate is a structured electronic document that provides tangible, auditable evidence that a source-code module has a specified property. Certificates are associated with program modules by encrypted links that resist forgery. Many forms of evidence can be accommodated. These can include the testimony of expert reviewers, results of testing, formal proofs of properties and

doi:10.1016/j.entcs.2007.02.045

software model checking. Different forms of evidence are supported by a variety of certificate servers, and may evoke varying degrees of trust in the certification provided.

Plover is one of the Programatica certificate servers. It is intended to provide a degree of assurance based upon the soundness of automated reasoning in a formal logic. Ultimately, the quality of this reasoning depends upon the correctness of the Plover tool itself, thus may be less convincing than if it supplied a formal proof tree. On the other hand, the reasoning conducted by Plover is fully automated, and is thus obtained with far less user expertise and expenditure of effort than is required for proof construction with the aid of a theorem-proving assistant.

Furthermore, Plover specifically implements reasoning in P-logic, which is the verification logic of Haskell98, whereas few other available proof assistants directly support a verification logic so closely tied to a wide-spectrum programming language.

1.1 The role of strategies

The architecture of a program verifier lends itself to the basic paradigm of term-rewriting. A rewriting system is an attractive way to formalize both the reduction rules of a computational language and the inference rules of a logical system. However, in a pure rewriting system there is no programmed control to select the locus of rewriting; a rewrite of any subterm that matches the left-hand side of a rule can occur. If a system is not confluent, a sequential implementation of undirected rewriting may fail to reach a desired normal form, even when such a form exists. For a confluent system, undirected rewriting can diverge, even when a normal form could be reached by a directed sequence of rewrites.

Conditional rewriting provides a degree of control to an otherwise undirected term-rewriting system by incorporating side conditions for a rule to fire, in addition to matching a redex. However, the side conditions that can be specified in a conditional rewrite only specify relations on local variables bound in matching the pattern of the rule to a redex. They cannot take into account information from the context of a possible redex, nor from the computational context, or "state", of a possible rule application. Thus it is difficult to specify policies such as top-down or bottom-up traversals of a term by conditional rewriting alone.

A further generalization of control in a rewriting system is provided by programmed *strategies*. Strategies incorporate conditional rewriting rules but also allow *sequencing* of rewriting steps and can specify *alternative* steps to be tried either non-deterministically or in a prescribed order [1] . Further generalization allows strategies to be parameterized over other strategies and to be recursively defined. Strategy parameters can carry information from the context surrounding a potential redex, to condition the application of a rewrite rule.

[1] Strategies that specify sequencing and alternatives have long been used in interactive theorem-provers [22,26,21] to reduce the workload of a human user.

1.2 Roadmap

This paper is about the use of generalized, programmed strategies to construct Plover. Section 2 introduces P-logic, a programming logic for Haskell. This is followed by an overview of the architecture of Plover in Section 3 and its implementation in Section 4, which includes a brief introduction to the strategy programming language Stratego. Normalization strategies are discussed in Section 5 and some type-specific strategies are given in Section 6. Section 7 contains a discussion of generic strategies, some of which are specific to Plover and some of which are not. The paper concludes with a summary of what has been accomplished to date in Section 8 and a survey of related work in Section 9.

2 A verification logic for Haskell

P-logic is specific to Haskell, meaning that the object-language terms whose properties can be asserted are expressions in Haskell98, well-typed in the context of the program module in which assertions appear. The semantic interpretation of predicates in P-logic is derived from a denotational semantics for Haskell98, enabling so-called total correctness assertions about any legal Haskell98 program to be formulated in P-logic.

Other examples of language-specific verification logics are ACL2 [18], a verification logic for Common Lisp, and Sparkle [10], a verifier for Clean 2.0. The advantage offered by direct formulation of assertions in a language-specific verification logic is that it is unnecessary to translate expressions and their asserted properties into some other logical formalism, which may have a different type system, and with the attendant risk that errors may be introduced in the translation.

2.1 Predicates refine types

Every predicate form definable in P-logic is subject to a typing discipline: a predicate is the refinement of a Haskell type. In fact, P-logic predicates themselves are typed in a Haskell type system extended with a distinguished type constant, Prop. A unary predicate P over Haskell expressions of type τ obtains the extended Haskell type $\tau \to \mathsf{Prop}$. A k-ary predicate is typed as a curried function, $\tau_1 \to \cdots \to \tau_k \to \mathsf{Prop}$.

P-logic provides basic constructions for unary predicates that are analogous to the constructors of Haskell types: the arrow (`->`), finite tupling, predicate constructor application and predicate disjunction, which is analogous to the sum-of-constructions by which data types are defined. Additional predicate constructions go beyond the constructions of Haskell types. These include predicate disjunction [2], predicate negation [3], predicate abstraction, least and greatest fixed-point constructions, and comprehensions that utilize formulas with quantified object variables in

[2] Predicate disjunctions are analogous to intersection types, which are not part of Haskell's type system.

[3] Predicate negation has no direct analogy as a type constructor, but is given a meaning in *P-logic* [19] that is compatible with the domain structure of a type frame.

the specification of a predicate.

Data constructors are implicitly lifted to predicate constructors. A data constructor typed as $C :: \tau_1 \to \cdots \to \tau_k \to \tau$ becomes a predicate constructor typed as $C :: (\tau_1 \to \mathsf{Prop}) \to \cdots \to (\tau_k \to \mathsf{Prop}) \to \tau \to \mathsf{Prop}$. A predicate, $C\,P_1 \cdots P_k$ is satisfied by a Haskell expression that semantically reduces to a head normal form $C\,e_1 \cdots e_k$, provided that each of the argument expressions e_i satisfies its respective predicate, P_i.

There is a distinguished, polymorphically typed, binary predicate $(\,\texttt{===}\,) :: \alpha \to \alpha \to \mathsf{Prop}$ that is interpreted as semantic equality of expressions. For more detail on the forms and meanings of predicates in P-logic, the reader is referred to [19].

2.2 The modality of well-definedness

Recall that in Haskell semantics, every type frame is a pointed cpo and furthermore, arrow, product and sum (data) [4] types are "lifted" above an undefined element that might not be expected in their categorical counterparts.

In consequence of this semantic structure, every unary predicate in P-logic is satisfied by the bottom (*undefined*) element in the type frame of its argument. Consequently, an assertion that an expression has an unannotated property, P, corresponds to a so-called *partial-correctness* assertion of the property that is intuitively understood by P. To express a total-correctness assertion, the symbol (\$) can be prefixed to a unary predicate expression to specify that it is *not* satisfied by a semantically undefined expression. We call this the *strong modality* of P-logic.

2.3 Fixed-point predicates

A unary (or monadic) predicate can be specified as a fixed-point of a predicate abstraction expression. Both least and greatest fixed-point definitions are possible, and in general, these definitions will differ semantically. A fixed-point predicate definition is prefaced by either of two binders of a predicate variable, `Lfp` or `Gfp`.

Of particular interest are the simplest, polymorphic fixed-point predicates,

$$\textbf{property}\ \mathit{UnDef} = \texttt{Lfp}\,X.\,X$$

$$\textbf{property}\ \mathit{Univ} = \texttt{Gfp}\,X.\,X$$

These define, respectively, a predicate satisfied only by a semantically *undefined* expression at each type and a predicate satisfied by every well-formed expression at each type. Fixed-point predicate definitions are particularly useful in defining properties of potentially infinite data structures, but that topic is beyond the scope of the present paper.

[4] Strictly speaking, a data type is a lifted sum only when its data constructors are not annotated in its declaration. If a data constructor is given a strictness annotation at some argument type, then the bottom element of that type frame is "coalesced" with the bottom element of the data type.

2.4 Integrating assertions with Haskell modules

In Haskell, the notion of a program module is a collection of function, type and class definitions together with a header that defines the external visibility of names defined in the module and the internal visibility of names that may be imported from other modules. A name used within a module may be qualified with the name of the module in which it is defined or may be unqualified, if no ambiguity results.

Programatica observes the name visibility conventions of Haskell modules, while extending the space of defined names to include the names of properties and assertions, whose definitions are enclosed within comment brackets `{-P: ... }` so that they will be ignored by an ordinary Haskell compiler.

A property or assertion definition can appear (inside Programatica comment brackets) at any program point at which a definition of a Haskell variable, type or class could appear. Lexical scoping rules determine the names in scope over a property or assertion definition, just as for ordinary definitions.

3 Towards automating program verification

The principal drawback of a language-specific verification logic is that tools for computer-aided verification in the logic may not exist and thus must be built. Three avenues of approach to constructing verification tools suggest themselves:

(i) Formulate a complete theory of the intended programming language as a library of proof rules in the syntax of a logical framework such as HL, Coq, or Isabelle, for which theorem-proving infrastructure already exists.

(ii) Compile programming language expressions and predicates of the verification logic into terms of a denotational semantics, expressed in a domain for which a theory has previously been formulated within an existing logical framework. This constitutes a *deep embedding* of the programming language into the internal language of the theory-enriched logic.

(iii) Program a custom verifier for assertions formalized in a language-specific verification logic.

The first approach presents a formidable task. To the best of our knowledge, it has not been attempted for any modern, wide-spectrum programming language. We have not given serious consideration to pursuing option (i) for *P*-logic.

The second approach is feasible, but results in a proof assistant that may be unintuitive to a programmer, as it supports reasoning in terms of the semantic representation of a program to be verified, rather than its surface syntax. We have adopted the third approach for two principal reasons.

- Reasoning, and the delivery of feedback to a user who is attempting to verify a conjectured property of a source-language program, is in terms of the program and the assertion that she has written. It has a better chance of being comprehensible to a programmer than does diagnostic feedback given in terms of a formal semantic representation.

- A custom verifier can, in principle, be fully automated, whereas the proof assistance tools of existing logical frameworks are designed for interaction with a human user. We shall say more on this topic later.

3.1 Verification is not theorem-proving

Often, program verification is confused with computer-aided theorem-proving, to which it is related. There are important differences as well. In mathematics, the proof of a theorem is often of as much interest as the theorem itself. In computer-aided theorem proving, the objective is to construct a proof object that can be displayed in a form intelligible to an knowledgable human. In contrast, when verifying a property of a program, the reasoning steps that are successfully discharged are of little or no interest so long as one is confident that they are logically sound. The interest is in the result, or in case of a failed proof attempt, in gaining an intuitive understanding of why the attempt failed.

Pushing the point further, in the Curry-Howard types-as-propositions analogy, a proof that a type is inhabited (proposition is satisfiable) is manifested as a program of the type. Discovery of a proof synthesizes such a program. Program verification, however, is analogous to proof checking, rather than proof synthesis. In verification, one is given a program conjectured to satisfy an asserted predicate. The verification task is to check that the truth of that conjecture follows from rules of logic and the formal theory of the programming language.

This distinction between theorem-proving and program verification has consequences for the design of a verification tool. Most importantly, since constructing an intelligible proof object is not an objective of verification, a verifier can take shortcuts that would be inadmissible in a theorem-prover, so long as they are coherent with the rules of the programming logic. Shortcuts include model-relative decision procedures for decidable subtheories of the programming logic. Such meta-strategies can dramatically improve the performance of an automated verification tool, relative to that of a theorem-prover.

3.2 The architecture of a verifier

In this section, we give an overview of the architecture of Plover—a custom verifier for P-logic. Plover operates as a verification server for Programatica. Plover relies upon the Programatica front-end tool, `pfe`, to parse, type-check and analyze dependencies of a Haskell module, generating input for Plover in the form of a list of terms in an abstract syntax for Haskell. `pfe` is run in a directory containing a number of Haskell source-code modules. It also has access to one or more Haskell libraries containing definitions that the source code may refer to.

3.2.1 Assertions as sequents

An assertion in P-logic is a propositional form in the context of a Haskell module. Sequents afford a representation of assertions that is particularly convenient for

Sequent	—	terms representing sequents
Prop	—	propositional terms
Pred	—	predicate terms
HTerm	—	Haskell expressions
HPattern	—	Haskell pattern forms
HType	—	Haskell type expressions

Fig. 1. Sorts of terms used in Plover

computational manipulation. The form of a Plover sequent is:

$$\mathcal{E}, \Gamma \vdash \Delta$$

where $\mathcal{E}$ is a finite set of type bindings (the *type environment*) for object variables that are in scope; Γ is a finite set of (implicitly conjoined) propositional forms, called the *assumptions* of the sequent and Δ is a finite set of (implicitly disjoined) propositional forms, called the *conclusions* of the sequent. Finite sets are represented as lists but the order of listing is of no logical consequence. To economize on the exposition in this paper, we shall ignore the type bindings in a sequent.

The general form of a Programatica assertion is a proposition in prenex normal form. Typings of the free and quantifier-bound variables have been calculated by `pfe`. To translate an assertion into sequent form, typings of these variables are collected in a type environment after skolemizing existentially quantified variables. If the quantifier-free matrix of an assertion is an implication, the set of implicands constitutes the list of assumptions in its sequent representation. Finally, if the implicant is a disjunction of propositions, the disjuncts constitute the set of possible conclusions; otherwise the entire implicant is listed as a singleton set of conclusions.

3.2.2 Inference rules

An inference rule of P-logic relates a consequent assertion to a finite set of antecedent assertions, called the *verification conditions* for the consequent. A rule is sound if logical validity of the antecedents entails validity of the consequent. A rule with an empty set of antecedents is an axiom.

Inference rules can be coded as term-rewriting rules. Given that a sequent is coded as abstract syntax terms, an inference rule can be implemented as a rewrite from its consequent to a list of antecedents.

3.2.3 Terms of several sorts

Rewrite rules may be triggered by matching on redexes that involve terms of several sorts (see Fig. 1). Rewrites that depend only on terms of sort Sequent correspond to structural rules of sequent calculus. Rewrites that depend on terms of sorts Prop

- A *pattern* is a strategy that binds its variables to components of a matching term, extending the current environment.
- A *term-builder* is a form that constructs a new term, using variable bindings from the current environment.
- A *rewrite rule* is a strategy consisting of a pattern followed by a term-builder.
- A *conditional* strategy is a rewrite rule scoped over an auxiliary strategy. The rewrite succeeds only if its auxiliary strategy succeeds in the current environment as extended by pattern-matching of the rule.
- A *sequential* composition of strategies (;) executes two strategies in sequence. It succeeds only if both components succeed.
- A *choice* composition of strategies succeeds if either one of its components succeeds. A choice can be nondeterministic (+) or left-biased (<+).

Fig. 2. Elements of a strategy

or Pred implement rules of propositional and predicate calculus that underlie *P*-logic. Rewrites that depend also on terms of sorts HTerm and HPattern implement rules that interpret Haskell semantics. A few rules analyze terms of sort HType to distinguish instances of Haskell type classes.

4 The Plover verification engine

4.1 Stratego: a strategy implementation language

Plover is implemented in a strategy programming language—Stratego [29]. In Stratego, the notion of strategy is formalized as a first-class computational entity, just as are functions or procedures in most programming languages. The subject to which a strategy is applied is a term; the environment for a strategy application is a set of bindings of variables that may occur in terms.

A strategy application may succeed or fail. If it succeeds, it produces a new term and may extend the set of extant bindings. If it fails, the extant term and bindings are unchanged.

The elements of strategy composition are summarized in Fig. 2. In Stratego, strategy definitions can be abstracted on strategy parameters and can be recursive.

Stratego is a compiled language whose runtime system is built on top of the A-term library [28]. This system provides maximal sharing of terms, implemented by internal hashing. In effect, it implements a content-addressable store, indexed by terms themselves.

4.1.1 Persistent tables

The Stratego library includes "wrapper" strategies that offer direct access to methods of the A-term library. Most significantly, there are strategies that build and access term-indexed tables, to provide persistent data storage. Persistent tables are

```
Eliminate-the-negatives :
    Consequence(assumptions,conclusions) ->
       Consequence(<conc>(neg_conclusions,pos_assumptions),
                   <conc>(neg_assumptions,pos_conclusions))
    where <map(try(\ Neg(Neg(p)) -> p \ ))> assumptions
                                            => assumptions';
          <filter(not(Neg(id)))> assumptions'
                                            => pos_assumptions;
          <filter(\ Neg(p) -> p\ )> assumptions'
                                            => neg_assumptions;
          <filter(not(Neg(id));substVar)> conclusions
                                            => pos_conclusions;
          <filter(\ Neg(p) -> p\ ;substVar)> conclusions
                                            => neg_conclusions
```

Fig. 3. Rewriting a sequent to remove negated propositions

used by Plover to record static information extracted from top-level Haskell definitions. Dynamic data summarizing statically scoped declarations from a Haskell module are stored in tables managed by a stack discipline. These tables provide an environment of non-local bindings that can be accessed by strategies.

4.2 Proof discovery by term rewriting

Plover implements proof rules of P-logic as rewrites, transforming a goal term of sort *Sequent* into one or more sequents representing its verification conditions. This process continues until every verification condition can be discharged, either by recognizing it to be an instance of an axiom of P-logic or by recognizing its conclusion to be a reflexive equality or an instance of one of its assumptions.

Proof rules of the logic are augmented by other rewrites that implement structural rules of sequent calculus. One of these, illustrated below, transforms a sequent that contains negated propositions among either its assumptions or its possible conclusions into a logically equivalent one containing only positive propositions.

4.2.1 Eliminating negated propositions from a sequent

Classical sequent calculus has the following bi-directional rules:

$$\frac{\Gamma,\neg P \vdash \Delta}{\Gamma \vdash P, \Delta} \qquad\qquad \frac{\Gamma \vdash \neg Q, \Delta}{\Gamma, Q \vdash \Delta}$$

Fig. 3 displays the code of a strategy that implements the two rules given above. The strategy has the form of a conditional rewriting rule. However, the auxiliary rules (the conditions) never fail, but instead define lists of propositions that are catenated to form lists of assumptions and of possible conclusions for the result sequent.

In reading the code, a strategy enclosed in angle brackets is applied to the term that follows, rather than to the extant "current term". The operator (`=>`) matches the term returned by a strategy appearing to the left of the operator with a pattern on its right. In its uses here, it is analogous to an assignment to a local variable.

The identifiers `conc`, `map`(1)[5] and `filter`(1) are the names of Stratego library strategies that implement functions. These functions calculate list concatenation, map an argument strategy over a list of terms and filter a list with respect to an argument strategy, respectively. The library strategy `not`(1) invokes its argument strategy on the current term, and in case it succeeds, the `not`(1) strategy fails. If the argument strategy fails, then `not`(1) succeeds and leaves the current term and binding environment unchanged.

`Eliminate-the-negatives` can be used either before or after a sequent has been rewritten by a strategy that propagates assumed equalities throughout a sequent. This strategy, described in Section 4.3.1, populates a persistent table implementing a mapping of terms to variables, that we call *env*. The strategy `substVar` is defined in Plover. It uses the mapping, *env*, to substitute in the term to which it is applied. It replaces every occurrence of a subterm that is in the domain of *env* with a variable.

4.2.2 Disjunctive assumptions and conjunctive conclusions

A set of assumptions is implicitly conjoined, so when an individual assumption has the explicit form of a conjunction, its conjuncts are simply listed as individual assumptions. A disjunctive assumption is a different proposition, however. The intuitive meaning of a sequent with a disjunctive assumption is that of an implication with alternative implicands. A strategy for simplifying such a sequent rewrites it into a set of nearly identical copies, each differing from the original sequent in that the disjunctive assumption has been replaced by one of its disjuncts. Each of the sequents in this set must be discharged to achieve discharge of the original sequent.

A conjunctive form in a conclusion has a similar interpretation. When a conclusion involves a conjunction, all of the conjuncts must be entailed by the assumptions in order to discharge the sequent. A sequent with a conjunctive conclusion is rewritten into a new set of sequents in which the conjunctive conclusion of the original sequent has been replaced by one of the conjuncts.

4.3 Strategies + Decision procedures = Verifier

While the fundamental strategy used in an automatic verifier is term rewriting, by augmenting rewriting with cooperating decision procedures for some decidable sub-theories of the programming logic, computational performance can be improved while economizing the number of rules that must be programmed.

A decision procedure for a specific theory utilizes computation in a model that provides a more compact representation for objects of the theory than does their

[5] Notation: The number in parentheses is not part of the strategy name, but indicates the arity of a strategy constructor, i.e. the number of strategy parameters that it takes.

representation as terms in a free algebra. When the union of two disjoint theories is decidable, they must have a common model[6]. Decision procedures that have a common model are said to *cooperate*. However, it is not always easy to find a common model for two theories, even when one exists.

Furthermore, in a verifier, we need the decision procedures for each sub-theory to cooperate with the rewriting-based, semi-decision procedure for assertions in the programming logic. Accordingly, we seek an injective map from abstract syntax terms in each sub-theory to terms in a model that may be used by a decision procedure. An injective map allows inversion (up to equivalence in the theory) of the transformation from a representation in the free algebra of terms to a more specific model. This supports interaction between a process of deduction by rewriting and model-relative computation. Plover's decision procedure implementation is derived from the Nelson-Oppen method for combining decision procedures [20].

4.3.1 A decision procedure for assumed equalities

The first decision procedure embedded in the Plover verifier interprets term equalities that appear in the assumptions of a sequent, by mapping equal terms to a unique representative of their equivalence class. To illustrate the implementation of a decision procedure, we shall outline the steps.

An assumed term equality is manifested by an abstract syntax term of sort Prop with the form $Equals\,(t_1, t_2)$, where t_1 and t_2 are understood to be meta-variables ranging over terms of sort HTerm. The following steps construct a model for deciding term equalities modulo the assumed equalities in a list of assumptions.

(i) **Map significant terms to fresh variables.** *Significant terms* in a list of assumed propositions are those of sort HTerm which are not simply representations of Haskell program variables ($HVar(\_)$ terms) and that occur as an argument of a propositional constructor or of the HTerm-sorted constructor *HApp* in a subterm of a significant term. Plover generates a fresh identifier for each significant term with a call to the Stratego library strategy `new` and extends the equality environment mapping, *env*, by mapping the term to the fresh identifier.

(ii) **Rewrite the assumptions, using the environment mapping.** The propositions assumed in the sequent are rewritten, replacing each occurrence of a significant term by $HVar(x_i)$, where x_i is the identifier to which the term is mapped by *env*. Every assumed equality now has the form

$$Equals\,(HVar\,(x_i), HVar\,(x_j))$$

(iii) **Orient the assumed equalities.** The arguments of each *Equals* term are oriented, using information gleaned from the inverse *env* mapping. To orient $Equals\,(HVar\,(x_i), HVar\,(x_j))$, determine whether the inverse mapping of either of x_i or x_j represents a term in weak head normal form. If so, choose that

[6] Disjointness of theories is a requirement of the Nelson-Oppen method. It is not a necessary condition for decision procedures to cooperate. Shostak's procedure for combining decision procedures can reconcile the overlap of constructors from distinct theories in some cases [25] but has not been incorporated into Plover.

variable to be the rightmost in the equality. If not, or if both are in normal form, the choice is determined by the lexical order of the identifiers themselves.

(iv) **Calculate the congruence closure of the equalities.** After orientation, the assumed equalities constitute a relation that is assymmetric apart from possible reflexive elements. An algorithm such as union-find [8] calculates the congruence closure of the relation, relating each of the identifiers to a unique representative of its equivalence class. Note that as a result of the orientation, if any of the original, significant terms in the equivalence class had been in normal form, one such term will be the inverse of the unique representative identifier.

Plover records the congruence relation in a persistent table for efficient reference.

(v) **Rewrite the conclusions of the sequent.** The possible conclusions are rewritten using the *env* mapping to replace signficant terms by their *HVar* equivalents, and using the congruence relation to replace the arguments of *HVar* terms by the representatives of their respective equivalence classes. All assumed equalities have now been propagated throughout the conclusions of the sequent.

4.3.2 *Equality propagation—an example*

Consider the sequent below, which is presented in Programatica source form:

$$\{a\} \texttt{===} \{(u,v)\},\ \{a\} \texttt{===} \{b\}\ \vdash\ \{(\lambda(x,y) \texttt{->} x)\, b\} \texttt{===} \{u\}$$

Plover will apply the steps of the decision procedure for equality.

Step i: The assumptions contain one significant expression, (u, v). This is bound to a fresh variable in the *env* mapping,

$$env:\ ((u,v), x_1)$$

Step ii: After rewriting the assumptions under the mapping, *env*, the sequent becomes

$$\{a\} \texttt{===} \{x_1\},\ \{a\} \texttt{===} \{b\}\ \vdash\ \{(\lambda(x,y) \texttt{->} x)\, b\} \texttt{===} \{u\}$$

Step iii: The assumed equalities are oriented. Since only the variable x_1 has a normal-form term as its inverse mapping under *env*, this is the only variable on which orientation is mandated. The oriented equations are:

$$\{a\} \texttt{===} \{x_1\},\ \{a\} \texttt{===} \{b\}$$

Step iv: Taking the congruence closure of the equations yields a single equivalence class. The orientation criterion determines the selection of a unique representative:

$$(\{a, b, x_1\}, x_1)$$

Step v: After substituting the equivalence class representative for equivalent variables throughout the sequent, it becomes:

$$\{x_1\} \texttt{===} \{x_1\},\ \{x_1\} \texttt{===} \{x_1\}\ \vdash\ \{(\lambda(x,y) \texttt{->} x)\, x_1\} \texttt{===} \{u\}$$

Now, however, when the inverse *env* mapping is used to replace occurrences of variables in the conclusion by equivalent, significant expressions, the sequent becomes:

$$\{x_1\} \texttt{===} \{x_1\},\ \{x_1\} \texttt{===} \{x_1\} \vdash \{(\lambda(x,y) \texttt{->} x)\ (u,v)\} \texttt{===} \{u\}$$

The *env*-inversion strategy is enabled in Plover when it attempts to reduce an application whose rator is a patterned abstraction and whose rand is a variable. In this case, *env*-inversion has yielded a sequent whose conclusion contains a reducible expression.

When Plover applies a reduction strategy, the result becomes

$$\{x_1\} \texttt{===} \{x_1\},\ \{x_1\} \texttt{===} \{x_1\} \vdash \{u\} \texttt{===} \{u\}$$

This sequent can be discharged immediately by recognizing that the conclusion is a reflexive equality.

4.3.3 Other decision procedures

A number of other well-known decision procedures can, in principle, be incorporated in a verifier.

Partial order modeling

Instances of Haskell's *Ord* type class define comparison operators, (<), (<=), (>), (>=). For the derived instances of this class, which include types *Int*, *Integer*, *Float*, *Char* and *String*, the comparison operators are interpreted as *Bool*-valued inequality comparisons. Assumed inequalities can be propagated by modeling partial orders, analogous to the modeling of equalities by congruence classes that was discussed in the previous section. A decision procedure for partial order has been implemented in Plover. It is used to resolve the values of some *Bool*-typed expressions involving derived instances of the *Ord* class.

Well-definedness of an expression

A particularly simple decision procedure is available for the strength (well-definedness) property of expressions that are not specified by recursion. Strength properties assumed of variables are recorded in a persistent table. The recorded properties incorporate as much as is known of the strictness properties of function-typed variables and variables of structured data types. For instance, the strength property assumed of derived instances of the operator (+) is represented by the predicate $\$(\$\mathit{Univ} \to \$(\$\mathit{Univ} \to \$\mathit{Univ}))$, characterizing a well-defined function that is total when applied to well-defined arguments. (It does not characterize the function as strict, however, as it does not specify its behavior when applied to an *undefined* argument.)

When the strength of an expression is needed to discharge a verification condition, a traversal strategy calculates it from the strength properties of its components. Strength properties calculated for a significant expression (one in the domain of the *env* mapping) are entered in the strength table, indexed by the equivalence class representative of the expression.

```
sorts Exp
constructors
   Var  :  String -> Exp
   Abs  :  String * Exp -> Exp
   App  :  Exp * Exp -> Exp
   Let  :  [(String * Exp)] * Exp -> Exp
```

(The constructor `Let` is used only to represent explicit substitutions.)

Fig. 4. Abstract syntax constructors for lambda calculus

Linear arithmetic

Equivalence of *Integer*-typed expressions with Pressburger (linear) arithmetic is decidable. While a decision procedure for this theory has not yet been implemented in Plover, it could be quite useful, particularly in simplifying *Bool*-typed terms that compare *Integer* subexpressions. Although the complexity of Pressburger-equivalence is daunting in the worst case, the occurrence of complicated, linear arithmetic formulas in Haskell programs is rare.

Positive rational arithmetic

There are also known decision procedures for linear rational arithmetic with equality and inequality comparisons. However, these procedures are not so easily integrated with other, useful decision procedures, as they involve transforming terms to representations in less transparent models. Rational arithmetic expressions are widely used in programs that use arrays as a primary data structure, but Haskell programmers tend to prefer other representations. It is unlikely that Plover will incorporate a decision procedure for rational arithmetic in the foreseeable future.

5 Strategies for normalizing terms

Plover is, of course, capable of reducing Haskell expressions to normal forms by applying reduction rules compatible with Haskell's denotational semantics. Strategies for normalization are so important to the success of a verifier that we shall spend some time discussing them. However, Haskell expressions have so many possible forms, including **let**, **case**, **if-then-else** expressions, records and data constructions, in addition to abstractions and applications, that we shall illustrate normalization strategies with a much simpler language—untyped lambda calculus with only the β-rule. Fig. 4 gives constructors for an abstract syntax of this language, in Stratego notation.

```
strategies
   whnf = Abs(id,id) + rec r(Var(id) + App(r,id))

   hnf = rec s(Abs(id,id) + rec r(Var(id) + App(r,s)))

   snf = rec s(Abs(id,s) + rec r(Var(id) + App(r,s)))
```

Fig. 5. Recognition strategies for normal forms

5.1 Recognizing normal forms

The most common definition of a normal form for the lambda calculus is that reached by exhaustive application of its reduction rules at every possible redex. That's not the only useful definition, however. If reduction is suspended under abstractions, then the form reached is called a *head* normal form and if it is, in addition, suspended in the *rand* term of an application, exhaustive reduction stops at a *weak* head normal form. Let's see how each of these can be characterized with a simple recognition strategy.

A recognition strategy is like a pattern, but since it does not need to produce bindings of the pattern variables, it may be defined recursively, to match terms of arbitrary depth. Three such definitions are given in Fig. 5, each corresponding to one of the three specifications of normal forms for the lambda calculus that were mentioned in the preceding paragraph. The definitions are subject to the assumption that all terms are well-sorted, i.e. that a constructor is only applied to subterms of the sorts given in the constructor's signature.

Let's examine the first definition in detail. It consists of two alternative strategy components, the second of which has a recursive definition scoping over two alternatives. Notice the use of the data constructors as strategy constructors. A data constructor, when lifted to become a strategy constructor, is satisfied by a term built with the same data constructor and whose argument terms satisfy the respective strategy arguments given to the strategy constructor. Note also the use of the `id` strategy as an argument to a lifted strategy constructor. `id` is a library strategy that always succeeds, leaving the current term and bindings unchanged. It is analogous to a wildcard designator in a pattern.

The first alternative of the `whnf` strategy is satisfied by an `Abs` construction with any well-sorted subterms as arguments. The recursively defined alternative is satisfied by any `Var` term and also by an `App` term whose rator is either a `Var` term or an `App` term in weak head normal form. Thus the recognition strategy excludes any `App` terms that has an `Abs` term as rator. There is no restriction on the rand subterm of an `App` construction.

The second definition is similar to the first, but adds the restriction that the rand of an `App` term must be in head normal form. Since the allowed forms of the rator and rand subterms differ, an additional level of recursive definition is needed to accommodate both forms. Finally, the third definition adds the restriction that the body of an `Abs` term must have the specified normal form, as well.

```
strategies
  Beta = \ App(Abs(Var(x),m),n) -> Let([(x,n')],m)
           where <RenameBoundVarsIn> n => n'              \

  LetElim = rec r({
              \ Let([],n) -> n                            \
            + \ Let([elmt | bindings],n) -> <r> Let(bindings,n')
                where <Replace> (elmt,n) => n'            \
            })
  Replace = rec r({
              \ ((x,m),Var(x)) -> m                       \
            + \ (elmt,App(m,n)) -> App(<r>(elmt,m),<r>(elmt,n)) \
            + \ ((x,m),Abs(y,n)) -> Abs(y,<r>(elmt,n))
                where <not(eq)> (x,y)                     \
           <+ \ (_,e) -> e                                \
            })
```

Fig. 6. Reduction strategies with explicit substitution

5.2 *Reduction strategies*

Reduction rules are readily programmed as alternative, conditional rewrites. Fig. 6 gives a Stratego encoding of rules for β-reduction with explicit substitution and capture-avoiding substitution. In Stratego notation, a rewrite rule is bracketed between backslash symbols.

In the syntax of a rule, a pattern appears to the left of the rewrite symbol (`->`) and to its right is a term-building strategy. If the rule is conditional, the condition strategy follows the keyword `where`. When a condition strategy succeeds it returns a result term. (A "pure" condition strategy occurs in the third alternative rule of the definition `Replace`. It is not followed by a pattern extension and is executed only to determine success or failure.)

When a strategy is enclosed in curly brackets, recursive invocations of the strategy bind fresh variables in auxiliary rule definitions, such as `m'` and `n'` in the definitions of `Beta` and `LetElim` above. Otherwise, bound variables would remain in scope throughout recursive invocations. Since a variable, once bound in a definition, cannot be rebound to a different term, failure to use curly brackets often results in failure of a recursively-defined strategy.

The strategy `RenameBoundVarsIn`, whose definition is not shown here, replaces every occurrence of the variable name bound in an `Abs` term with a fresh name. `RenameBoundVarsIn` uses a package of generic renaming strategies supplied in the Stratego library, specializing them to terms of sort `Exp`.

The pattern `[`*hd* `|` *tl*`]` matches a non-nil list, binding the pattern variable *hd* to the head of the list and *tl* to its tail. The library strategy `eq` is satisfied by a pair of syntactically identical terms.

Notice that the last alternative listed in the definition of `Replace` is separated

```
strategies
  BetaSubst = Beta; LetElim

  Lazy-eval = rec r(whnf <+ App(r,id); try(BetaSubst; r))

  Eager-eval = rec r(hnf <+ App(r,r); try(BetaSubst; r))

  Strong-eval = rec r(snf <+ Abs(id,r)
                         + App(r,r); try(BetaSubst; r))
```

in which `try`(1) is a library strategy defined as

```
  try(s) = s <+ id
```

which never fails but executes the strategy `s` if it can succeed.

Fig. 7. Three strategies for normalization

by the operator symbol (`<+`) rather than the symbol (`+`). The symbol (`<+`) designates left-biased choice rather than nondeterministic choice of alternative strategies. Since the pattern of the final alternative would match an arbitrary pair of terms, it overlaps the patterns of each of the rules that precede it, and is programmed to fire only as a default alternative.

5.3 Normalization strategies

Finally, we are ready to present strategies to normalize lambda expressions, using the β-rule and explicit substitution. Three normalization strategies, corresponding to the three normal forms presented earlier, are shown in Fig. 7. Notice that these strategies are defined using lifted constructors, as were the recognition strategies, rather than rewrite rules, as were the reduction strategies.

Each normalization strategy first tries the recognition strategy for its respective normal form, returning immediately in case the current term is normalized. Otherwise, if the current term matches an `App` construction, then `BetaSubst` is tried after normalizing appropriate subterms of the construction. In case `BetaSubst` succeeds, it is still not assured that the result term is normalized, thus the normalization strategy is applied recursively to the result.

Plover incorporates strategies similar to `Lazy-eval` and `Eager-eval` but for Haskell terms. Both lazy and strict normalization strategies are used repeatedly by Plover and interact with a partial decision procedure for semantic equality to simplify Haskell terms.

6 Type-specific strategies

Many of the strategies used in Plover are designed to simplify expressions or assertions that are specific to a Haskell type, or type constructor. This section explains a few of these and suggests how they might be generalized.

6.1 Structure splitting

Some types uniquely determine the top-level structure of normal-form terms of the type. Product types (finite tuples) and data types with only a single constructor have this property. Structure-determining types see greater use in Haskell programs than in many other languages.

When the argument of an application, the definiens of a local definition or the scrutinee of a case expression has a structure-determining type, it can be assumed to be equal to a head normal form of the type, subject to the auxiliary verification condition that the given expression of the type is well defined. Since the constructor of a head normal form for a structure-determining type is unique, the assumed equality can be formalized as a new assumption. To synthesize a head normal form term, the unique data constructor is applied to a fresh variable at each of its argument positions.

This strategy, which we call *structure splitting*, is used by Plover whenever it enables further strategies of application reduction, **case** reduction, or **let** reduction (inlining of local definitions) to succeed.

6.2 Boolean reduction

Plover implements a library of small strategies that realize a theory of *Bool*-typed expressions, a boolean algebra over the Haskell Prelude-defined operators `(&&)`, `(||)` and `not`. This library also includes strategies that interpret the Prelude-defined comparison operators `(<)`, `(<=)`, `(>)` and `(>=)`, using a decision procedure for partial order.

One could implement a complete decision procedure for boolean satisfiability to attempt to resolve values for *Bool*-typed expressions. Plover does not do this. Instead, it implements a series of weaker strategies that attempt to eliminate the most common use made of *Bool*-typed expressions: as discriminators in **if-then-else** expressions and guards.

6.3 If-then-else elimination

There are three basic techniques that may be used to eliminate an **if-then-else** (*Ite*) expression when it occurs in the conclusion of a sequent:

(i) reduce the *Bool*-typed discriminator of an *Ite* to a constant by a reduction strategy, thus allowing the *Ite* to be reduced;

(ii) deduce a constant value for the discriminator as a consequence of the assumptions, which again, allows reduction of the *Ite*;

(iii) split the sequent into two, one in which the discriminator of the *Ite* expression has been assumed equal to *True* and a second in which it is assumed equal to *False*. Verify the logical validity of both sequents.

Each successive strategy option subsumes the ones that precede it; however, each is potentially more computationally expensive than are its predecessors, so all

```
 IteElim(disch) =
  \ ((HIte(b,_,_),assumptions),bvs) -> e
    where
      ?ite_bvs;
      IteElimThen => (e1,strengthAssertion,trueAssertion);
      <disch> strengthAssertion;
      ( <disch> trueAssertion;  !e1
      + !ite_bvs; IteElimElse => (e2,falseAssertion)
        <disch> falseAssertion; !e2
      ) => e                                               \

 IteElimThen =
  \ ((HIte(b,e1,_),assumptions),bvs) ->
      (e1,Consequence(assumptions,[Has(b,Strong(Univ))]),
          Consequence(assumptions,bprop))
    where <free-vars(\ HVar(x) -> [x]\ ,FringeAs)> b => fvs;
          <isect> (bvs,fvs); [];
          // the intersection of bound and free variables is empty
          <Bool-to-prop-list> b => bprop                   \

 IteElimElse =
  \ ((HIte(b,_,e2),assumptions),bvs) ->
       (e2,Consequence(assumptions,bprop))
    where <Bool-to-prop-list> HApp(HVar("not"(b))) => bprop \
```

Fig. 8. A strategy for if-then-else elimination

three are useful. We shall examine the second and third options, respectively named `IteElim` and `IteSplit`.

6.3.1 IteElim

The `IteElim` strategy, slightly simplified for the sake of exposition, is defined by a conditional rewrite, is given in Stratego code in Fig. 8.

Plover uses `IteElim` as a strategy argument to a bottom-up term-traversal strategy (not shown here) on a Haskell expression occuring as an argument of an *P*-logic predicate in the conclusion of a sequent. It tries `IteElim` on every subterm. The traversal strategy also propagates a list of properties assumed in the hypotheses of the sequent and the list of Haskell variables bound in the context surrounding each subterm. These are supplied, along with the subterm itself, in a three-tuple to which `IteElim` is applied.

`IteElim` uses two subordinate strategies, `IteElimThen` and `IteElimElse`, in alternatives that try to discharge either the proposition that the *Bool*-typed discriminator of an *Ite* expression equals *True* or that it equals *False*, in the context in which it is found. If either of these hypotheses is verified, then `IteElim` succeeds, returning the expression on the corresponding arm of the *Ite* expression. Otherwise, it fails.

A discharge strategy is passed as a parameter to `IteElim`. It is used to discharge the side condition that the discriminator of the *Ite* is well-defined, before attempting to discharge one or the other of the hypotheses about the value of the discriminator.

6.3.2 IteSplit

When `IteElim` fails to eliminate an *Ite* expression, Plover can try structure splitting on the two possible values of its discriminator. The strategy `IteSplit` implements the following inference rule of *P*-logic:

$$\frac{\begin{array}{c}\Gamma \vdash b ::: \$\mathit{Univ} \\ \Gamma,\, b \mathrel{===} \{\mathit{True}\} \vdash e_1 ::: P \\ \Gamma,\, b \mathrel{===} \{\mathit{False}\} \vdash e_2 ::: P\end{array}}{\Gamma \vdash \{\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2\} ::: P}$$

`IteSplit` is a relatively weak strategy because it does not examine all possible bindings of the free variables that occur in the discriminator expression, but only the possible values of the expression itself. A more refined strategy employing *SAT*-solving could improve `IteSplit`, but might be too computationally expensive for the occasional benefit it would provide.

`IteSplit` also has the potential to be computationally expensive, as it generates three verification conditions that must be discharged for it to succeed. When *Ite* expressions are nested, the number of verification conditions to be discharged by this strategy increases exponentially with the nesting depth.

6.4 Case elimination

Just as for *Ite* expressions, the first strategy to try to eliminate a **case** expression in the conclusion of a sequent is evaluation of the case scrutinee by reduction. If the scrutinee reduces to a head normal form, some number of the possible matches with patterns on the case branches can be eliminated, allowing the case expression to be rewritten into a simpler one that is semantically equivalent in the context of the sequent. If the **case** rewrites to one with only a single branch, it can be further rewritten into an explicit abstraction, derived from the remaining case branch, and applied to the scrutinee. The synthesized application can then be reduced by a rule specializing β-reduction to Haskell's semantics.

A generalization of `IteSplit` to data types other than *Bool* appears quite promising for eliminating **case** expressions. Before splitting the scrutinee, analysis is required to determine that the patterns of a **case** expression completely cover the constructions of their type. When case branches involve guards, branch coverage analysis requires the `IteElim` and `IteSplit` strategies to be extended to guards, to resolve their possible valuations.

If the branches of a **case** expression are complete with respect to possible values of the scrutinee, a sequent in which the **case** occurs can be split into a number of sequents, one per case branch. Each of the derived sequents will contain a structure-splitting assumption that the scrutinee matches a specific pattern form. An additional verification condition asserts well-definedness of the scrutinee.

6.5 *Eliminating equalities of abstractions*

When the conclusion of a sequent asserts an equality between an explicit abstraction and another expression (necessarily of the same type), the abstraction can be eliminated by applying both sides of the asserted equality to a common, fresh variable. If the abstraction pattern happens to be of a structure-determining type, structure splitting may enable immediate reduction of the synthetic application.

7 Generic strategies

Several strategies employed in Plover cannot be called type-specific, yet they deal with program constructions that are particular to Haskell. These include strategies for local definitions introduced in **let** or **where** clauses, guarded expressions, fixed-point induction and strategies that instantiate quantified assumptions. We shall consider in detail the strategies Plover uses for **let** expressions and for instantiating lemmas.

7.1 *Strategies for let expressions*

A **let** expression constitutes a list of local definitions that scope over a single object expression. The order in which definitions are listed is semantically unimportant in Haskell, as they scope over one another, as well. Thus a set of definitions may be mutually recursive.

A **let** expression can be simplified by manifesting the equalities entailed by its definitions, whenever possible. Several strategies for **let** expressions and local definitions are implemented in Plover.

Rename variables bound in the pattern of a definition.

The strategy that renames with fresh variables all occurrences of variables bound in the pattern of a local definition cannot fail. This strategy prepares a non-recursive definition to be lifted into a surrounding context.

Simplify definitions by structure matching.

When the left side of a definition is a structured pattern (i.e., not a simple variable) that is matched by the right hand side, the definition can be split into one or more, simpler definitions by structure matching. This may enable subsequent definition inlining.

Since the definitions in a **let** expression use lazy matching, structure matching succeeds only if the entire pattern on the left of a definition is matched by the expression on the right.

Lift local definitions into an enclosing scope.

A definition can be lifted from a **let** expression into an enclosing scope if its right hand side contains no occurrence of a variable bound in the pattern of another definition of the **let** expression.

Sort local definitions in order of their dependency.

Since the order of definitions in a common scope has no semantic significance, a list of definitions can be order-sorted into a sequence consistent with the partial order of their dependencies. In case the list contains a clique of mutually recursive definitions, the relative order of the clique is preserved by this sorting.

Inline simple variable definitions.

A non-recursive definition whose left-hand side pattern is a simple variable can be inlined by capture-avoiding substitution of its right-hand side for every free occurrence of the variable in the scope of the **let**.

Eliminate redundant let forms.

A **let** expression, all of whose definitions have been eliminated by inlining and/or lifting to a surrounding scope, is redundant and can be rewritten to its object expression alone.

7.2 A strategy for asserted equality of let expressions

Let's consider an example in which Plover uses the strategies for **let** expressions described above to discharge an asserted equality. The example is taken from a module (shown in Fig. 9) that implements an instance of the *Monad* class in Haskell. The assertion of one of the monad laws relates two expressions that after inlining function definitions, contain embedded **let** expressions. It displays many of the intricacies, apart from recursive definitions, that can occur in relating such expressions.

To simplify the exposition of the example, we have ignored verification conditions that require strong (i.e. well-definedness) properties of terms. In a full proof of the asserted laws, Plover uses strength induction, which is not discussed in this paper.

Consider the assertion `M_Assoc` from Fig. 9. After substituting the definition of

```
module State

where

newtype State s a = ST (s -> (a, s))

-- instance Monad (State s) where

return x      = ST (s -> (x, s))

(ST c)>>=f  = ST (s -> let (x, s')  = c s
                            ST c'  = f x
                        in c' s')

{-P : -- monad laws

assert M_Id1 =

   All f, x.

      {f} ::: $(Univ -> $(ST $Univ)) ==>

         {return x >>=f} === {f x}

assert M_Id2 =

   All m. {m>>=return} === {m}

assert M_Assoc =

   All f, g, m.

      {(m>>=f)>>=g} === {m>>=(x -> f x >>=g)}

-}
```

Haskell's *Monad* class specifies that an instance must define two functions, $return :: a \to m\ a$, and $(\texttt{>>=}) :: m\ a \to (a \to m\ b) \to m\ b$, where m represents the monad type constructor.

In the code above, the newtype declaration specifies a representation by functions from a type of state objects to the product of a parameter type and the state type. The data constructor, ST, has no semantic significance.

The function *return* injects its argument into the structure of the monad representation. Its definition shows that the function specified by $return\,x$ does not change the state component.

The operator (>>=) defines function application in the monad. An application m >>= f produces a function that evaluates the monadic structure of m by applying it to a state variable to produce a (value, state) pair. The components of this pair are passed as arguments to f, which returns a new (value, state) pair.

Fig. 9. A simple state monad in Haskell with asserted laws

the *bind* operator (>>=), the asserted equality becomes:

$$\left\{\begin{array}{l}
\textbf{let}\ ST\,d_1 = \textbf{let}\ ST\,c_1 = m \\
\qquad\qquad\quad \textbf{in}\ ST\,(\lambda s_1\ \texttt{->}\ \textbf{let}\ (x_1, s_1') = c_1\ s_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ST\,c_1' = f\ x_1 \\
\qquad\qquad\qquad\qquad\qquad\ \ \textbf{in}\ c_1'\ s_1') \\
\textbf{in}\ ST\,(\lambda r_1\ \texttt{->}\ \textbf{let}\ (y_1, r_1') = d_1\ r_1 \\
\qquad\qquad\qquad\quad\ ST\,d_1' = g\ y_1 \\
\qquad\qquad\quad\ \textbf{in}\ d_1'\ r_1')
\end{array}\right\}\ \texttt{===}$$

$$\left\{\begin{array}{l}
\textbf{let}\ ST\,c_2 = m \\
\textbf{in}\quad ST\,(\lambda s_2\ \texttt{->}\ \textbf{let}\ (x_2, s_2') = c_2\ s_2 \\
\qquad\qquad\qquad\qquad\quad ST\,c_2' = \textbf{let}\ ST\,d_2 = f\ x_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\quad ST\,(\lambda r_2\ \texttt{->}\ \textbf{let}\ (y_2, r_2') = d_2\ r_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ST\,d_2' = g\ y_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textbf{in}\ d_2'\ r_2') \\
\qquad\qquad\quad \textbf{in}\ c_2'\ s_2')
\end{array}\right\}$$

Step 1: Dependency-order sorting of definitions.

To implement a test for equivalence of two **let** expressions in a common context of assumptions, Plover first sorts the list of definitions in each by order of their dependency. In our example, the original ordering of definitions within each **let** expression is consistent with their dependency.

Step 1': Opportunistic inlining.

Plover tries to inline any local definition whose left side is a single variable, then attempts to reduce terms at which a substitution for the variable was successful. This strategy fails if no reduction is possible, leaving the original term unchanged.

Step2: Structure matching.

Before attempting further analysis of a **let** expression, Plover simplifies its non-recursive definitions by attempting structure matching on each. In the example, there are no possibilities for structure matching of the original definition forms.

Step 3: Lift independent definitions into a surrounding scope.

Steps 1 and 2 can be embedded in a traversal strategy to reorganize and simplify **let** expressions throughout a possibly larger term. During a term traversal, each remaining definition that is both non-recursive and independent of any definition that precedes it in the list ordering is analyzed to determine whether it depends on any variable bound in an immediately enclosing a **case** branch or abstraction term. If it does not, the enclosing expression is rewritten to a **let** expression into which the independent definition has been syntactically lifted.

Step 4: Eliminate redundant let forms.

If, after independent definitions are lifted, a nested **let** expression is left with an empty list of definitions, the **let** construction is eliminated, leaving only its object expression.

Applying Steps 3 and 4 to the example, two definitions can be lifted one level: the definition $ST\ c_1 = m$ from the first nested **let** on the left hand side of the equality and the definition $ST\,d_2 = f\ y_2$ from the second nested **let** on the right hand side. Both of the nested **let** expressions collapse after lifting the definitions and the asserted equality becomes:

$$\left\{\begin{array}{l}
\textbf{let } ST\,c_1 = m \\
\quad ST\,d_1 = ST\,(\lambda s_1\, \texttt{->}\, \textbf{let } (x_1, s_1') = c_1\ s_1 \\
\qquad\qquad\qquad\qquad\quad ST\,c_1' = f\ x_1 \\
\qquad\qquad\qquad\qquad \textbf{in } c_1'\ s_1') \\
\textbf{in } ST\,(\lambda r_1\, \texttt{->}\, \textbf{let } (y_1, r_1') = d_1\ r_1 \\
\qquad\qquad\qquad ST\,d_1' = g\ y_1 \\
\qquad\qquad \textbf{in } d_1'\ r_1')
\end{array}\right\} \texttt{===}$$

$$\left\{\begin{array}{l}
\textbf{let } ST\,c_2 = m \\
\textbf{in } \ ST\,(\lambda s_2\, \texttt{->}\, \textbf{let } (x_2, s_2') = c_2\ s_2 \\
\qquad\qquad\qquad\qquad ST\,d_2 = f\ x_2 \\
\qquad\qquad\qquad\qquad ST\,c_2' = ST\,(\lambda r_2\, \texttt{->}\, \textbf{let } (y_2, r_2') = d_2\ r_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ST\,d_2' = g\ y_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } d_2'\ r_2') \\
\qquad\qquad\quad \textbf{in } c_2'\ s_2')
\end{array}\right\}$$

When Step 5 (see below) is applied to the top-level **let** definitions on both sides of the asserted equality, the right hand sides of the first definitions in each list are found to be equivalent and the patterns on their left sides match. Thus it can be assumed that the corresponding pattern variables are equal. The matching definitions are eliminated from the **let** expression and the equality $(c_1 \texttt{===} c_2)$ is appended to the assumptions of the sequent in which the assertion is embedded.

Steps (1-4) change the structure of a term and can reveal opportunities for reduction strategies to be applied. If reduction further simplifies the right hand side of a local definition, the entire series of steps is repeated until no further simplification is possible.

In the example, structure matching now applies to simplify the second definition of the top-level **let** expression on the left of the asserted equality and the third definition of the first nested **let** on the right side. After simplification, the patterns of the resulting definitions become single variables and the right sides are explicit abstraction expressions. Step 1' succeeds in inlining these definitions and reducing the resulting applications. The asserted equality then becomes:

$$\left\{\begin{array}{l}
ST\,(\lambda r_1\, \texttt{->}\, \textbf{let } (y_1, r_1') = \textbf{let } (x_1, s_1') = c_1\ r_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad ST\,c_1' = f\ x_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } c_1'\ s_1' \\
\qquad\qquad\qquad ST\,d_1' = g\ y_1 \\
\qquad\qquad \textbf{in } d_1'\ r_1')
\end{array}\right\} \texttt{===} \left\{\begin{array}{l}
ST\,(\lambda s_2\, \texttt{->}\, \textbf{let } (x_2, s_2') = c_2\ s_2 \\
\qquad\qquad\qquad\quad ST\,d_2 = f\ x_2 \\
\qquad\qquad\quad \textbf{in let } (y_2, r_2') = d_2\ s_2' \\
\qquad\qquad\qquad\quad ST\,d_2' = g\ y_2 \\
\qquad\qquad\quad \textbf{in } d_2'\ r_2')
\end{array}\right\}$$

Repeating Steps 3 and 4 collapses the second nested **let** expression in the terms on each side of the asserted equality, yielding

$$\left\{\begin{array}{l}
ST\,(\lambda r_1\, \texttt{->}\, \textbf{let } (x_1, s_1') = c_1\ r_1 \\
\qquad\qquad\qquad\quad ST\,c_1' = f\ x_1 \\
\qquad\qquad\qquad\quad (y_1, r_1') = \ c_1'\ s_1' \\
\qquad\qquad\qquad\quad ST\,d_1' = g\ y_1 \\
\qquad\qquad\quad \textbf{in } d_1'\ r_1')
\end{array}\right\} \texttt{===} \left\{\begin{array}{l}
ST\,(\lambda s_2\, \texttt{->}\, \textbf{let } (x_2, s_2') = c_2\ s_2 \\
\qquad\qquad\qquad\qquad ST\,d_2 = f\ x_2 \\
\qquad\qquad\qquad\qquad (y_2, r_2') = d_2\ s_2' \\
\qquad\qquad\qquad\qquad ST\,d_2' = g\ y_2 \\
\qquad\qquad\quad \textbf{in in } d_2'\ r_2')
\end{array}\right\}$$

Structure matching of the two sides of the asserted equality reduces it to an equality of explicit abstractions. Upon applying these abstractions to a common, fresh variable (the strategy of Section 6.5), we obtain:

$$\left\{\begin{array}{l}\mathbf{let}\ (x_1, s_1') = c_1\, v_1\\ \qquad ST\, c_1' = f\, x_1\\ \qquad (y_1, r_1') = \ c_1'\, s_1'\\ \qquad ST\, d_1' = g\, y_1\\ \mathbf{in}\ d_1'\, r_1'\end{array}\right\} === \left\{\begin{array}{l}\mathbf{let}\ (x_2, s_2') = c_2\, v_1\\ \qquad ST\, d_2 = f\, x_2\\ \qquad (y_2, r_2') = d_2\, s_2'\\ \qquad ST\, d_2' = g\, y_2\\ \mathbf{in}\ d_2'\, r_2'\end{array}\right\}$$

Step 5: Eliminate corresponding, equivalent definitions.

Consider the remaining, non-recursive definitions of each **let** expression that is to be compared for equality. Each forms a sublist of definitions whose order is compatible with the partial order of their dependency. A heuristic strategy tries to match the sublists extracted from the **let** expressions of the original equality assertion by postulating equalities of their respective right hand sides, and attempting to discharge the hypothesized assertions.

If a definition at the head of one list fails a test of equivalence with the corresponding element of the other list, it is moved back in list order, respecting the partial order of dependency, and the equivalence test is repeated with the next definition in order. If a definition from one list cannot be proved equivalent to any definition in the other list, the entire equality strategy fails.

When a definition is found equivalent to one in the opposite list, corresponding pattern-bound variables of the equivalent definitions are assumed equal, augmenting the set of assumptions of the sequent, and the equivalent definitions are removed. Elimination of equivalent definitions continues until the non-recursive components of both **let** expressions have been eliminated. Equivalence of the residual expressions, relative to the augmented list of assumptions, is then scrutinized by other strategies.

Completing our example, repeated application of Step 5 is now able to determine equivalences of all corresponding definitions on the left and right hand sides of the asserted equality. Replacing each pair of equivalent definitions by a set of equalities appended to the assumptions of the sequent transforms the sequent with original assumptions Γ to:

$$\Gamma,\ c_1 === c_2,\ x_1 === x_2,\ s_1' === s_2',\ c_1' === d_2,\ y_1 === y_2,\ r_1' === r_2',\ d_1' === d_2' \vdash \{d_1'\, r_1'\} === \{d_2'\, r_2'\}$$

This sequent is discharged after its assumed equalities have been propagated to the conclusion.

This five-step strategy, while it may appear complicated, has proven to be effective in a number of examples.

7.2.1 Recursive definitions in **let** *expressions*

To discharge conjectured equivalences between residual lists of recursive, local definitions, strategies similar to the five-step procedure described above for non-recursive definitions can be used. Significant differences are that Step 1' (inlining) is skipped, and in Step 4, an entire set of mutually recursive definitions must be lifted out of a surrounding syntactical context as a block.

Step 5 must also be modified for recursive definitions. A strategy that tests the equivalence of two lists of mutually recursive definitions must tentatively assume equalities between the variables bound in the patterns of corresponding definitions from each of the lists. (This is an instance of *assume-guarantee* reasoning [17].) Such a strategy is not complicated, but it has not yet been implemented in Plover.

7.3 Induction strategies

The next group of strategies to be implemented in Plover will support several forms of induction. None of these has been implemented at the time of this writing, however.

7.4 Lemmas and their instances

In verification, as in theorem-proving, a task can often be simplified by breaking it into stages, utilizing assertions previously verified (or assumed) as lemmas from which to prove an assertion. The form taken by a lemma is typically that of a proposition, universally quantified over some number of its variables. To incorporate a such lemma, it is written as an implicand of the assertion to be verified.

When a quantified proposition appears as an implicand in an assertion, translating the assertion into sequent form places the quantified formula among the set of assumptions of the sequent. A quantified formula cannot be used directly to discharge the conclusion of sequent. However, it is logically sound to generate one or more instances of a quantified assumption, appending each instance to the set of assumptions.

The question is, how should an assumed, quantified formula be instantiated so that it is likely to enable further progress towards discharging a sequent, while avoiding pollution of the set of assumptions with spurious instances? It is important to avoid generating spurious instances, for if the quantified proposition happens to be a disjunctive or implicative formula, its instances, appearing in a set of assumptions, will cause a sequent to be split into multiple sequents, each of which requires discharge.

7.4.1 An instantiation strategy for quantified implications

It would be easy to determine how to instantiate the quantified variables of a lemma if one could anticipate how the instance would be used in subsequent steps of a derivation. Two possibilities come to mind.

(i) If the matrix of a quantified assumption is an implication, then an instance might prove useful if its implicant matches the conclusion of the sequent. In this case, a possible verification condition would be a new sequent having the same set of assumptions as the original, but whose conclusion is a conjunction of the implicands of the matching instance of the quantified assumption.

(ii) Alternatively, a quantified implication could be useful if one or more of its implicands matches another assumed proposition. In this case, a new assumption could be generated by applying a rule of modus ponens to the previously assumed proposition and the matching instance of the implication.

Plover has the capability to rewrite a sequent containing a quantified assumption into one or more new sequents, using either of the strategies outlined above. Any one of the generated sequents can suffice as a verification condition to discharge the original sequent.

Additional instantiation strategies suggest themselves in case the matrix of an assumed proposition is an equality. (This is the form usually taken by algebraic laws.) Then a criterion for instantiation is that one side of the equality matches a term that occurs either in another assumption or the conclusion of the sequent. Since equality is symmetric, exhaustive application of an equality matching strategy might easily diverge. A strategy that assigns a direction to an assumed equality and uses it only for directed rewriting avoids an obvious cause for non-termination.

7.4.2 Example—Instantiating a quantified equality assumption

The following example is taken from a Haskell module (see Fig. 10) that models a simple hardware memory. In this model, the type of an address is left unspecified; it is only required to belong to the *Eq* class. This ensures that the comparison operator (==) :: $Eq\ a$ => a -> a -> Bool is defined for the type of addresses, but assumes no properties of the comparison operator.

Here, the expression *extend x y f a* is a functional specification of a memory mapping obtained by "storing" a value y at location x in a memory whose previous mapping function is f. The function *eqFor* is introduced as a ruse to ensure that the type instance of the *Eq* class used in typing (==) agrees with the type of the variable x that occurs in the expression *extend x y f x*, in the assertion *ExtendApply*. Otherwise, since the scope of a Haskell type variable is restricted to a single definition, the types given to distinct occurrences of (==) in elaborating the property definition *Reflexive* and the definition of *extend* would not be recognized as two occurrences of the same operator. (In the abstract syntax output by the `pfe` tool, these symbols would have qualified names. The qualifiers have been omitted here, for simplicity.)

Following elaboration of the defined identifiers in *ExtendApply* and symbolic reduction of the propositional expression *Reflexive* {*eqFor x*}, the assertion becomes:

```
All x, y, f.
  (All x'. {x'==x'} === {True}) ==>
  {x} ::: $Univ ==>
  {if x==x then y else f x} === {y}
```

In sequent form, this assertion appears as:

$$(\forall x'.\ \{x'{==}x'\} \mathrel{===} \{\mathit{True}\}),\ \{x\} ::: \$\mathit{Univ} \vdash \{\mathsf{if}\ x{==}x\ \mathsf{then}\ y\ \mathsf{else}\ f\ x\} \mathrel{===} \{y\}$$

A strategy for instantiating generalized assumptions attempts to match the expression on the left hand side of the quantified equality with subexpressions occuring in the conclusion or in another assumption. In this example, only one match is possible. The strategy appends an instance of the quantified formula to the list of

```
module Extend
where
 extend          ::  Eq a => a → b → (a → b) → (a → b)
 extend x y f  =  λa -> if a==x then y else f a

-- eqFor x specifies the instance of (==) at the type of x

eqFor          ::  Eq a => a → a → a → Bool
eqFor x y z  =  y==z

{-P:
property Reflexive = {| rel :: Eq a => a → a → Bool |
                        All x :: a. {x ‘rel‘ x} === {True} |}

assert ExtendApply = All x, y, f.
                     Reflexive {eqFor x} ==>
                     {x} ::: $Univ ==>
                     ({extend x y f x} === {y})
-}
```

Fig. 10. A Haskell module that models extension of a function

assumptions.

$$(\forall x'.\ \{x'\texttt{==}x'\}\ \texttt{===}\ \{\mathit{True}\}),\ \{x\}\ :::\ \$\mathit{Univ},\ \{x\texttt{==}x\}\ \texttt{===}\ \{\mathit{True}\}$$
$$\vdash\ \{\textsf{if}\ x\texttt{==}x\ \textsf{then}\ y\ \textsf{else}\ f\ x\}\ \texttt{===}\ \{y\}$$

Once the assumed reflexive interpretation of (==) has been made explicit, the equality propagation strategy outlined in Section 4.3.1, followed by **if-then-else** reduction, transforms the sequent into a form that can be discharged.

8 Summing up

We have described a strategy-driven approach to designing and implementing a language-specific, automatic program verifier. The major advantage of a language-specific verifier is that it provides a translation-free interface to source text written in an actual programming language. An obvious drawback is that it requires considerable effort to implement. However, alternative approaches also require considerable effort.

The tool described in this paper, Plover, is a verifier for Haskell98. Its implementation is a work in progress. This work has been facilitated by the choice of a well-designed strategy language (Stratego) for its implementation.

Strategies generalize the notion of conditional rewrite rules in an important way. While firing of a conditional rewrite rule depends upon conditions specified on components of a redex term, strategies may also encompass conditions that depend upon non-local data, i.e. on the context that surrounds a redex. The Stratego language provides an algebra of operators for combining strategies sequentially, alternatively and parametrically.

8.1 What has been accomplished to date?

Plover is currently able to verify properties of many Haskell programs that do not make use of recursion. (Strategies for recursion await implementation.) Examples include:

- Properties of fetch and store operations in a simple (functional) model of a virtual store with allocation and deallocation of address blocks;
- Associativity and commutativity of conjunction and disjunction operators in a boolean algebra over a diamond lattice of pairs of *Bool*-typed values;
- Monad laws (some of which have been weakened so as to be valid) for state, continuation and resumption monads programmed in Haskell.

8.1.1 Plover from a user's perspective

The current state of the Plover implementation is frustrating to its users. Although it is capable of verifying some interesting properties already, it offers a user inadequate feedback about why it fails on a verification attempt. There are two typical reasons that Plover fails:

(i) the assertion that it is asked to verify depends upon a Haskell language feature that Plover does not yet comprehend or for which suitable strategies are not yet in place, or

(ii) the assertion it has been given is not provable in *P*-logic and may not be valid for Haskell's semantics.

Additional attention by Plover's implementors should make its future releases more robust in identifying cases of (1) but it is unlikely ever to be fully self-aware of the limitations of its capabilities. Circumstance (2) may be remedied by displaying subgoals that Plover failed to discharge. While many failed subgoals are simply not germaine to a successful verification, some are, and an attentive user may learn to recognize these.

A technique that a user should try when a verification attempt fails is to formulate simpler subgoals and assert them as "lemmas" rather than assuming that Plover will discover them. Subgoal identification has long been a successful technique for simplifying a problem.

Once a user has had success in verifying properties of a program module, Plover's automation should greatly simplify regression-verification during subsequent evolution of the module.

8.2 Complexity issues

The potential complexity of theorem-proving has always been viewed as a potential limitation on the use of automation. In mathematics, the intuition of a knowlegeable and patient human can guide construction of a proof of a complex result. Such intuition is unavailable to an automated proof tool.

However, program verification is not theorem-proving, although many of the same techniques can be employed. In program verification, most conjectured propositions are not as deep as an interesting mathematical theorem and do not require similarly complex proof constructions. Nor are they sufficiently intriguing to a human to command her devoted attention for a long period of time. A result (yes or no), rather than the structure of a proof, is the primary objective of a verification task. Full automation seems the most desirable approach to program verification.

The computational complexity of a procedure is a measure of how many individual tasks it spawns, as a function of the size of its input. In program verification, there are two obvious candidates for complexity blowup. One is the number of alternatives to be considered in discharging an individual verification condition. This is determined by the "accidental" complexity of the programming language structure, i.e. how many ways there are to express a program. A second source of complexity is the generation of multiple secondary goals in the discharge of a verification condition, such as occurs in rewriting a sequent with disjunctive assumptions or a conjunctive conclusion (Section 4.2.2).

Experience with Plover has shown that the accidental complexity of programming constructions is of no real importance for the computational complexity of program verification. Even when many possibilities exist for the discharge of a verification condition, most of them are exhausted quickly; for instance, by pattern-match failures.

On the other hand, when a single strategy step can generate multiple verification conditions, the complexity of a verification can grow exponentially in the depth of its derivation tree. This is where real complexity problems lie. Realizing this, one programs strategies for a verifier with an eye to minimizing VC-multiplying situations. This is another area in which strategies offer a programming advantage over conventional, conditional rewriting techniques.

9 Related work

There is a large body of existing work on computer-aided verification and theorem proving. We cannot adequately survey it here, but merely mention several of the systems that have been developed to model and verify properties of programs or formal specifications.

9.1 Reflective term-rewriting systems

Systems with powerful capability for deduction have evolved from the term-rewriting paradigm. Conditional term-rewriting systems have been extended with capabilities for reflection, allowing strategies for manipulating both terms and rules to be written in the same language framework that is used to write programs and specifications. Two such systems are *ELAN* [1] and *Maude* [5]. These systems do not incorporate model-based decision procedures, but instead axiomatize decidable theories with rewrite rules.

Plover doesn't use reflection, having opted instead for an implementation in a strategy programming language separate from Haskell, the language whose terms are the subject of analysis.

9.2 Explicitly designed verifiers

Verification tools have been designed for several existing programming languages and some formal specification languages as well. Most verifiers integrate decision procedures with logical reasoning to discharge verification conditions derived from formal property assertions embedded in or appended to a program or specification. They do not construct proof terms to justify their conclusions. *Simplify* (see below) has been included in this summary because it is used as a deductive engine by several verifiers, although it is not, by itself, specialized to any programming language.

Sparkle.

Sparkle [10] is a verification assistant specially constructed for the functional programming language *Clean* [3,24]. It comprehends the syntax and semantics of *Clean 2.0*. *Sparkle* is an interactive tool, offering a user a library of proof tactics that may be invoked while attempting to prove an asserted property of a *Clean* program fragment.

As a property specification language, *Sparkle* uses a first-order predicate logic whose predicate former is equality and whose object terms are expressions in *Core-Clean*, extended with a notation for an undefined value ($\perp$), which is essential for reasoning about a language with default lazy evaluation semantics. The expressiveness of the *Sparkle* specification language is somewhat less than that of *P*-logic, which supports predicate abstraction and recursive predicate definitions.

ACL2.

ACL2 [18] is a verification assistant for the applicative fragment of Common Lisp. Its programming logic is based upon equational reasoning over Common Lisp terms, thus its logic is integrated with a programming language widely used for modeling systems. The inference engine underlying ACL2 is derived from the Boyer-Moore theorem prover [2]. It interprets equational theories as rewrite rules to transform terms.

ACL2 is a mature system that has been used to model and verify properties of a variety of systems, including complex processor subsystems. It provides a flexible user interface to support interactive proof construction. Proof strategies are programmed in Common Lisp and archived in theory books, along with previously proved theorems.

Plover has been inspired by the success of ACL2 but the complexities of Haskell, and hence, of P-logic, require additional strategies for verification.

Java verification

LOOP [27] is a prototype verifier for a fragment of Java. Developed at Nijmegen University, it uses PVS as an inference engine, providing it with a theory of Java semantics expressed in the PVS specification language, and a translation of an annotated Java program to be verified. It has been used to verify security properties of the API for Java Card, a “smart” electronic cash card.

ESC-Java [12] generates first-order formulas as verification conditions for properties asserted by embedding annotations in Java programs. It is based upon a partial theory of the semantics of Java expressions. It operates automatically, without interactive input from a human user. Verification conditions generated from an annotated Java program by *ESC-Java* are submitted to *Simplify*, a first-order theorem-prover.

If *Simplify* cannot discharge a verification condition generated by *ESC-Java*, it reports failure and constructs a possible counterexample to the assertion. Failure reports are intended to be evaluated by a human user to determine whether a suggested counterexample represents a real program bug or a false alarm resulting from the incompleteness of the theorem prover.

Simplify

Simplify [11] is the automatic, first-order prover used by verifiers *ESC-Java* and *ESC-Modula3*. It comprehends quantified as well as quantifier-free formulas. *Simplify* combines a complete decision procedure for the theory of equality (the Nelson-Oppen procedure) with ones for linear rational arithmetic and some incomplete procedures for linear integer arithmetic. It uses an incomplete, pattern-driven strategy to find relevant instances of assumptions that are given as quantified formulas.

Like many other first-order provers, *Simplify* tries to prove the validity of a formula, Q, by testing satisfiability of its negation, $\neg Q$. If it finds no satisfying assignment of truth values to the literals of $\neg Q$ that is also consistent with the underlying theories (positive rational arithmetic, partial orders) on which it is based, then it has proved the original formula. If it finds such an assignment, it reports it as a possible counterexample. (Since Plover does not use the SAT-solving approach, it is not able to generate counterexamples from sequents that it fails to discharge.)

To test for satisfiablility, *Simplify* puts a formula into disjunctive normal form, then backtracks over alternatives, searching for a satisfiable disjunct. In each conjunctive subformula, *Simplify*'s heuristic strategy for quantified subformulas tries to find an instance that matches one or more “trigger” terms in another formula. Promising instances are conjoined with the remaining conjuncts and the enriched clause is tested for consistency. Plover's strategy for generating relevant instances of a quantified assumption is similar.

PVS

PVS [22] is a custom-designed verifier for assertions formulated in the PVS specification language, which is derived from classical, typed higher-order logic. The designers of PVS have pioneered the inclusion of cooperating decision procedures in a verifier [9,25]. The verifier has an interactive interface that allows a user to specify strategies but is capable of carrying out proof steps automatically, using pre-programmed strategies. PVS has been used to specify and verify fault-tolerant flight control systems, secure computing platforms and other safety or security-critical systems.

Although the object language embedded in the PVS specification language has been defined to have a reduction semantics, there is no independent compiler for it, thus it is not used as a programming language for applications.

9.3 Theorem provers

Theorem provers are based upon a small core of logical rules that guarantee their soundness. All reasoning steps follow by application of one or more of these rules to the formal axioms of a specified theory or by application of previously proved lemmas concluded from that theory. A theorem prover does not rely on decision procedures and does not contain programmed interpretations of axioms of a user-specified theory in its trusted base.

A theorem prover can be based upon either a constructive or classical logic. However, most have utilized a constructive logic, or type theory, faithful to the philosophy of the Curry-Howard isomorphism.

NuPrl

One of the earliest, and still one of the most widely used of the theorem provers is *NuPrl* [6]. Its logic is based upon Martin-Löf type theory, extended with the Y combinator as a realizer, and incorporating a theory of Scott domains. Thus *NuPrl* is equipped to support program verification.

Extensive theory libraries have been developed for *NuPrl*, enabling it to be used to verify properties of a number of software specifications and even for implementations of some small systems. However, if she wishes to succeed in a non-trivial verification exercise, a human user should be thoroughly trained in the intricacies of *NuPrl*.

Other theorem provers that have been used in verification

Several other theorem provers have been adapted to verify properties (typically equivalences) of specifications formulated in a higher-order, typed term language comprehended by the theorem prover. These include *HOL* [13], *Isabelle* [21], *Coq* [26,7], *Agfa/Alfa* [15,14], *LF/Elf* [16,23] and *LEGO* [4]. Each of these systems is intended to function as a proof assistant to a human user, who steers an attempted proof discovery by directing the application of strategies (historically referred to as *tactics*). Some of these systems are equipped with sophisticated user interfaces that assist in keeping track of undischarged proof obligations and suggesting possibly useful strategies [7].

However, to use any of these systems on programs in an existing programming language, the program and its property assertion must be translated into semantic and logical equivalents, expressed in the term language supported by the prover.

References

[1] Peter Borovansky, Clause Kirchner, Helene Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[2] Robert S. Boyer and J. Strother Moore. A theorem prover for a computational logic. In M. Stickel, editor, *Proc. of 10th Internat. Conf. on Automated Deduction (CADE'1990)*, volume 449 of *LNCS*. Springer Verlag, 1990.

[3] Tom Brus, Marko van Eekelen, Maarten van Leer, and Rinus Plasmeijer. Clean: a language for functional graph rewriting. In *Proc. of the Third Internat. Conf. on Functional Programming Languages and Computer Architecture (FPCA'87)*, volume 274 of *LNCS*, pages 364–384. Springer Verlag, 1987.

[4] Rod Burstall. Computer assisted proof for mathematics: an introduction using the LEGO proof system. Technical Report ECS-LFCS-91-132, University of Edinburgh, 1990.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In *Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *LNCS*, pages 76–87. Springer Verlag, June 2003.

[6] Robert L. Constable et al. *Implementing Mathematics wit the NuPrl Development System.* Prentice-Hall, 1986.

[7] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Ed.* MIT Press and McGraw-Hill, 2001.

[9] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[10] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers—SPARKLE: A functional theorem prover. In *Proc. of 13th Internat. Workshop on Implementation of Functional Languages (IFL'01)*, volume 2312 of *LNCS*, pages 99–118. Springer Verlag, 2001.

[11] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[12] Cormac Flanagan, K. Rustan Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245. ACM Press, June 2002.

[13] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[14] Thomas Hallgren. Homepage of the proof editor Alfa. `http://www.md.chalmers.se/ hallgren/Alfa`, 2004.

[15] Thomas Hallgren and Aarne Ranta. An extensible proof editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer Verlag, 2000.

[7] The *Alfa* proof editor is particularly helpful in this regard.

[16] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *IEEE Symposium on Logic in Computer Science (LICS 1987)*, pages 194–204. IEEE Computer Society Press, June 1987.

[17] Thomas A. Henzinger, Marius Minea, and Vinahyak Prabhu. Assume-guarantee reasoning for hybrid systems. In *Proc. of the 4th Internat. Workshop on Hybrid Systems: Computation and Control (HSCC 2001)*, volume 2034 of *LNCS*, pages 275–290. Springer Verlag, 2001.

[18] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[19] Richard B. Kieburtz. P-logic: Property verification for Haskell programs. `ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/Plogic.pdf`, 2002.

[20] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[21] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th Internat. Conf. on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[23] Frank Pfenning. Logic programming in the LF logtical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[24] Rinus Plasmeijer and Marko van Eekelen. Clean Version 2.0 Language Report. `http://clean.cs.ru.nl/download/Clean20/doc/CleanRep2.0.pdf`, December 2001.

[25] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In *Proc. of 13th Internat. Conf. on Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Artificial Intelligence*, pages 1–18. Springer-Verlag, 2002.

[26] The *Logical* team. The Coq proof assistant. `http://coq.inria.fr`, 2006.

[27] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Software (TACAS'2001)*, volume 2031 of *LNCS*, pages 299–312. Springer Verlag, 2001.

[28] M. G. J. van den Brand, H. A. de Jong, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000.

[29] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.