An Overview of the Programatica Toolset

Thomas Hallgren James Hook Mark P. Jones
Richard B. Kieburtz
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Rd, Beaverton, Oregon, USA

http://www.cse.ogi.edu/PacSoft/projects/programatica/

1 Introduction

With ever increasing use of computers in safety and security critical applications, the need for trustworthy computer systems has never been greater. But how can such trust be established? For example, how can we be sure that our computer systems will not destroy or corrupt valuable data, compromise privacy, or trigger serious failures that could, in the worst cases, lead to loss of life?

Industry and academia have attacked this problem in many different ways, demonstrating concretely that the use of techniques such as systematic design processes, rigorous testing, or formal methods can each contribute significantly to increased reliability, security, and trustworthiness. There are obviously some significant differences between these techniques, but there is also a unifying feature: each one results in some tangible form of *evidence* that provides a basis for trust. Examples of such evidence include a record of the meeting where a code review was conducted, the set of test cases to which an application was subjected, or a formal proof that establishes the validity of a critical property.

Unfortunately, the diversity, volume, and variability of evidence that is needed in practice, even in the development of some fairly small systems, makes it hard to manage, maintain, and exploit this evidence as a project evolves and as meaningful levels of assurance are required. Without support, practical difficulties like these could easily discourage or prevent system builders from capturing and leveraging evidence to produce systems in which they, and their users, can develop a well-placed trust.

In the Programatica project at OGI [24], we are exploring the role that tools can play in facilitating and supporting effective use of evidence during the design, development and maintenance of complex software systems. From a programmer's perspective, Programatica provides a development environment for an extended version of the functional programming language Haskell [16] that provides a notation (and an associated logic, called *P-logic* [17]) for expressing properties of Haskell code. From a verification perspective, we expect that a broad spectrum of techniques will be useful in establishing such properties, including code review, testing, and formal methods. To support *evidence management*, Programatica provides: mechanisms for capturing different forms of evidence in *certificates*; methods exploiting fine-grained dependency analysis to automate the tasks of tracking, maintaining, and reestablishing evidence as a system evolves; and tools to help users understand, manage, and guide further development and validation efforts. One way to understand Programatica is as a framework for *Extreme Formal Methods*: we expect property assertions to be developed and established in parallel with the code that they document, just as test cases are developed in extreme programming [2].

This paper describes the current version of the Programatica tools and illustrates how they address some of the above goals in practice. The tools, while still a work in progress, can already manipulate Haskell programs in various ways and have some support for evidence management. In Section 2, we describe some of the features that the tools provide to support program development in Haskell, including facilities for browsing, slicing, and generating coding metrics. Section 3 describes functionality that is more directly targeted at the needs of evidence management. Programatica is designed to provide an open architecture for interfacing with external tools, and we illustrate this by describing two different 'server' components. The first, described in Section 3.1, connects Programatica programs and property assertions to Alfa, a proof editor for structured type theory. The second, described in Section 3.2, provides a corresponding connection between Programatica and Plover, an automatic verification engine for P-logic. Notes on the implementation are provided in Section 4.

2 Program Development Tools

2.1 Basic Command-line Tools

The functionality of our tools is available through a simple command line interface. Many different tasks can be performed by passing appropriate arguments to a program called `pfe` (the "Programatica Front-End"). Some of the functionality of `pfe` is also available through a graphical interface (Section 2.3).

The tools operate on a *project*, which is simply a collection of files containing Haskell source code. The command `pfe new` creates a new project, while `pfe add` adds files and `pfe remove` removes files from a project. There is also a command `pfe chase` to search for source files containing needed modules. This is *the only* command that assumes a relationship between module names and file names. A wrapper script `pfesetup` uses `pfe new` and `pfe chase` to create a project in a way that mimics how you would compile a program with `ghc --make` [9] or load it in Hugs [13].

Like batch compilers, `pfe` caches various information in files between runs (e.g., type information). `pfe` also caches the module dependency graph, while other systems parse source files to rediscover it every time they are run. This allows `pfe` to detect if something has changed in a fraction of a second, even for projects that contain hundreds of modules.

Once a project has been set up, the user can use `pfe` for various types of queries. For example, `pfe iface` M displays the interface of module M, `pfe find` x lists modules that export something called x and `pfe uses` $M.x$ lists all places that use the entity called x that is defined in Module M.

2.2 The HTML Renderer

The command `pfe webpages` generates web pages for a project, with one page per module. Every identifier in the generated HTML code is linked to its definition. Extensive tagging is used and a reference to a style sheet is made, allowing the user to customize the look of the resulting web pages. An example is show in Figure 1.

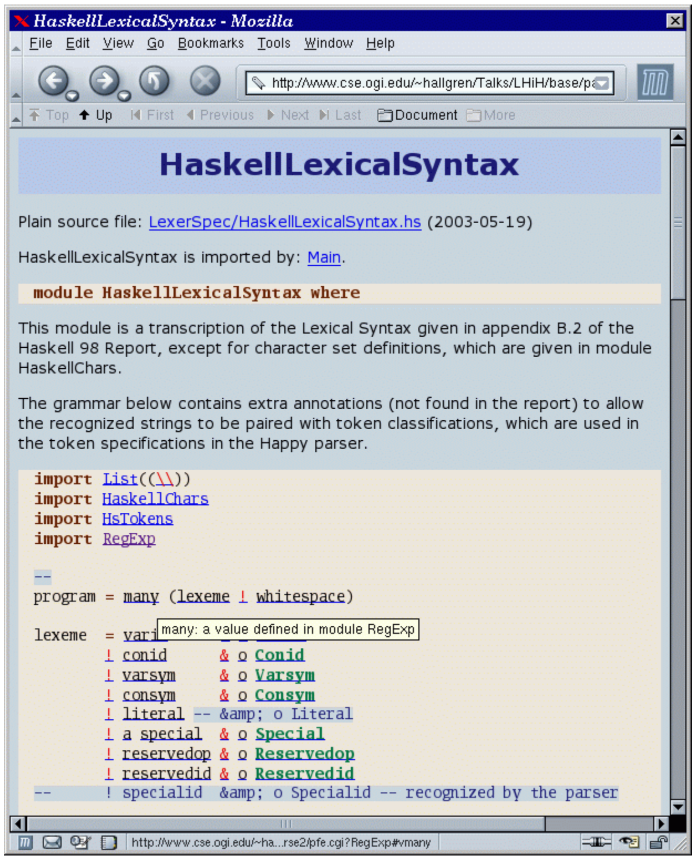
While the syntax highlighting provided by editors such as Emacs and Vim is based on a simple lexical analysis, our tool also makes use of syntax analysis to distinguish between type constructors and data constructors. Also, for hyperlinking identifiers to their definition, a full implementation of the module system and the scoping rules of Haskell are used. Although the source code is parsed, the HTML code is *not* generated by pretty-printing the abstract syntax tree, but by decorating the output from the first pass of our lexical analyzer [11], preserving layout and comments.

We also support a simple markup language for use in comments, keeping the plain ASCII presentation of source text readable, while, at the same time, making it possible to generate nice looking LaTeX and HTML from it. We used this system, for example, to prepare our paper on the Haskell Module System [8].

Our HTML renderer is a tool for documenting and presenting *source code*, in contrast to Haddock [18], which is a tool for producing user documentation for *libraries*.

2.3 The Haskell Source Code Browser

While HTML rendering makes it possible to use a standard web browser to browse Haskell programs, we have also implemented a dedicated Haskell browser (Figure 2). It assumes that a project has already been set up with `pfe`, and is started with the command `pfebrowser`. It provides a syntax highlighted and hyperlinked source view similar to the one provided in the HTML rendering, but


HaskellLexicalSyntax - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://www.cse.ogi.edu/~hallgren/Talks/LHiH/base/p
Top Up First Previous Next Last Document More

HaskellLexicalSyntax

Plain source file: LexerSpec/HaskellLexicalSyntax.hs (2003-05-19)

HaskellLexicalSyntax is imported by: Main.

```
module HaskellLexicalSyntax where
```

This module is a transcription of the Lexical Syntax given in appendix B.2 of the Haskell 98 Report, except for character set definitions, which are given in module HaskellChars.

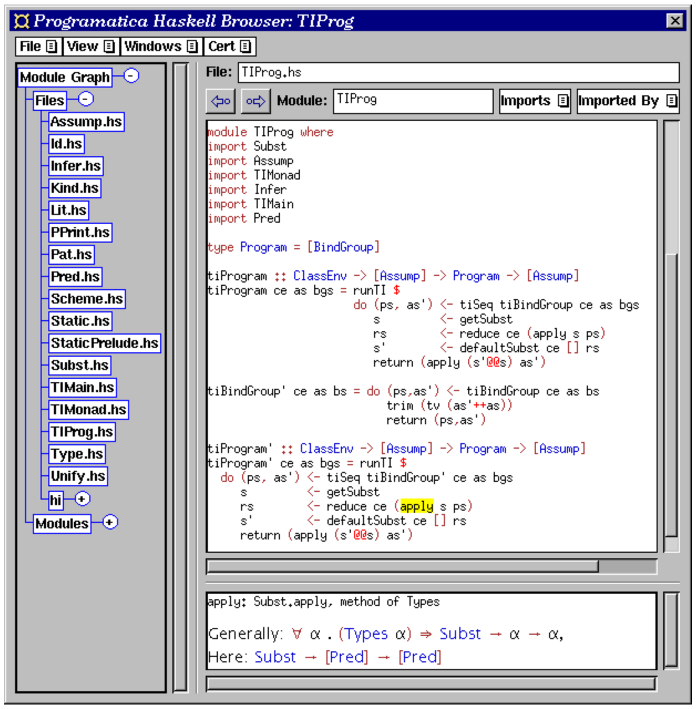
The grammar below contains extra annotations (not found in the report) to allow the recognized strings to be paired with token classifications, which are used in the token specifications in the Happy parser.

```
import List((\\))
import HaskellChars
import HsTokens
import RegExp

--
program = many (lexeme ! whitespace)

lexeme  = vari many: a value defined in module RegExp
        ! conid      & o Conid
        ! varsym     & o Varsym
        ! consym     & o Consym
        ! literal -- &amp; o Literal
        ! a special  & o Special
        ! reservedop & o Reservedop
        ! reservedid & o Reservedid
--      ! specialid  &amp; o Specialid -- recognized by the parser
```

Figure 1: A sample Haskell module rendered as HTML by the Programatica Tools.

it also has some additional functionality. For example, the user can click on an identifier to find out what its type is. The browser can reload and retypecheck a module after it has been edited. Information is cached internally to make this quick. Currently, reloading and retypechecking a moderately sized module (a few hundred lines or so), takes just a second or two.

At present, the tools type check complete modules, and do not provide any type information type checking fails. In the future, we might make the type checker more incremental, providing partial information in the presence of type errors. Other possible features for future development include turning the browser into a Haskell editor, and creating a dedicated proof tool for Haskell, perhaps similar to the Sparkle proof tool [7] for Clean.



```
module TIProg where
import Subst
import Assump
import TIMonad
import Infer
import TIMain
import Pred

type Program = [BindGroup]

tiProgram :: ClassEnv -> [Assump] -> Program -> [Assump]
tiProgram ce as bgs = runTI $
                      do (ps, as') <- tiSeq tiBindGroup ce as bgs
                         s         <- getSubst
                         rs        <- reduce ce (apply s ps)
                         s'        <- defaultSubst ce [] rs
                         return (apply (s'@@s) as')

tiBindGroup' ce as bs = do (ps,as') <- tiBindGroup ce as bs
                           trim (tv (as'++as))
                           return (ps,as')

tiProgram' :: ClassEnv -> [Assump] -> Program -> [Assump]
tiProgram' ce as bgs = runTI $
  do (ps, as') <- tiSeq tiBindGroup' ce as bgs
     s         <- getSubst
     rs        <- reduce ce (apply s ps)
     s'        <- defaultSubst ce [] rs
     return (apply (s'@@s) as')
```

```
apply: Subst.apply, method of Types

Generally: ∀ α . (Types α) ⇒ Subst → α → α,
Here: Subst → [Pred] → [Pred]
```

Figure 2: The Programatica Haskell Browser. The user has clicked on a use of `apply` to confirm that it applies a substitution to a list of predicates.

2.4 Program Slicing

Viewing a program as a set of definitions, the slicer computes, given an identifier, the subset of the definitions that are referenced, directly or indirectly, from that identifier. For example, by slicing with `Main.main` as the starting point, you can remove dead code from a program. This functionality is provided by the command `pfe slice` $M.x$. The output is a set of valid Haskell modules that, for example, can be loaded in Hugs. (It will not be a complete Haskell *program* if slicing starts from something other than `Main.main`.)

Slicing is driven by a dependency analysis on type-checked code, allowing unused instance declarations to be eliminated. However, to keep the slicer simple, top-level declarations are treated as atomic units and this can make the slicing coarse, for example including definitions of default methods (and everything they depend on) in class declarations, even if they are not needed in any instance.

The slicer preserves the module structure of the source program. Import and export declarations are adjusted appropriately.

Because of the annoying monomorphism restriction, eliminating a definition that *is not used* can change the meaning of a declaration that *is used.* Dependencies caused by this side effect are currently not taken into account by our slicer.

2.5 Software Metrics

The `pfe` program provides commands for calculating an ad-hoc selection of software metrics:

- `sizemetrics`: number of lines of source text per module (includes comments and blank lines)
- `locmetrics`: number of lines of code per module (excludes comments and blank lines)
- `importmetrics`: number of imports per module
- `exportmetrics`: number of exports (importers) per module
- `classmetrics`: number of instances per class

Support for other forms of metrics might be added in the future.

3 Evidence Management

At the moment, the command line tools and the browser support the creation and validation of certificates for the following basic forms of evidence:

- informal claims ("I say so"),
- random testing with QuickCheck [5],
- formal proof in Alfa [10], and
- automated proof with the dedicated P-logic verifier *Plover.*

There is work in progress on support for other forms of evidence, and our implementation has an extensible architecture to make it easy to plug them in using *certificate servers* to act as bridges between our code and external tools.

When source code is changed, the validity of existing certificates can be affected. Revalidating certificates might require some work by the user. To help identify those changes that can actually influence the validity of a certificate, dependencies are tracked on the definition level (rather than the module level). Changes are detected by comparing hash values computed from the abstract syntax. As a result, we avoid false alarms triggered by changes to irrelevant parts of a module, or by changes that only affect comments or the layout of code.

One of the main tasks of a certificate server is to translate P-logic property assertions and the corresponding parts of the program that they refer to into the language/notation of an external tool. (Slicing, as described in Section 2.4, is particularly useful here because it can help to reduce the amount of code that must be translated.) The following sections describe how this is accomplished in building servers for Alfa (Section 3.1) and Plover (Section 3.2).

3.1 Structured Type Theory in Alfa

The translation to Structured Type Theory [6] allows asserted properties to be proved formally in the proof editor Alfa [10]. The translation is based on type checked code. Polymorphic functions are turned into functions with explicit type parameters and overloading is handled by the usual dictionary translation, turning class declarations into record type declarations and instance declarations into record value definitions.

The syntax of Structured Type Theory is simpler than that of Haskell, so a number of simplifying program transformations are performed as part of the translation. This affects, for example, list comprehensions, the do-notation, pattern matching, derived instances and literals. There are also various ad-hoc transformations to work around syntactic restrictions and limitations of the type checker used in Alfa. Apart from this, the translated code looks fairly similar to the original Haskell code.

While Haskell has separate name spaces for types/classes, values and modules, Structured Type Theory has only one, so some name mangling is required to avoid name clashes. This is done in a context-independent way, so that when some Haskell code is modified, the translation of unmodified parts remains unchanged, allowing proofs about unmodified parts to be reused unchanged.

While type theory formally only deals with total functions over finite data structures, the translator allows any Haskell program to be translated. Alfa contains a syntactic termination checker [1] that can verify that a set of structurally recursive function definitions are total. When the translation falls outside that set, any correctness proofs constructed in Alfa entail only *partial correctness*, and we leave it to the user to judge the value of such proofs. In the future, we might use a more sophisticated termination checker and/or change the translation to use *domain predicates* [3] to make partiality explicit and to make the user responsible for providing termination proofs.

While the type system of plain Haskell 98 can be mapped in a straight-forward way to the *predicative* type system of Structured Type Theory, we foresee problems extending the translation to cover existential quantification in data types and perhaps also higher rank polymorphism.

```
-- Synchronous stream processors
module SPS where

newtype S i o = S (i->(o,S i o))
feed (S sps) i = sps i

runS _       []     = []
runS sps (i:is) = continue (feed sps i) is
continue c is = fst c:runS (snd c) is

(>*<) :: S i1 o1 -> S i2 o2 -> S (i1,i2) (o1,o2)
sps1 >*< sps2 = S sps
  where
    sps (i1,i2) = ((fst o1,fst o2),snd o1>*<snd o2)
       where o1 = feed sps1 i1
     o2 = feed sps2 i2

assert Separation1 = {-#cert:Separation1#-}
  All sps1 . All sps2 . All is1 . All is2 .
  {runS (sps1>*<sps2) (zip is1 is2)} === {zip (runS sps1 is1) (runS sps2 is2)}
```

Figure 3: The Haskell module `SPS`, defining synchronous stream processors and asserting a separation property.

3.1.1 Example: Using the Alfa Certificate Server

In this section we illustrate the practical details involved in creating evidence for an asserted property using the Programatica tools with the Alfa certificate server in their current state.

The source code in this example is a Haskell module `SPS` (see Figure 3), defining *synchronous stream processors* and asserting a simple separation property for parallel composition of synchronous stream processors. (c.f., [24])

The first step in applying the Programatica tools is to set up a project using `pfesetup`:

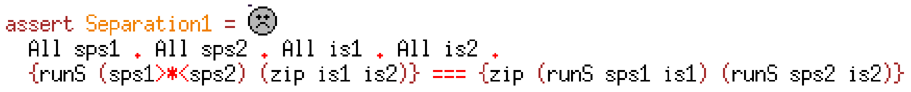
```
pfesetup SPS.hs
```

Next, we can start `pfebrowser` and take a look at the Haskell code for module SPS, noting in particular the following definitions,

- $S\ i\ o$: the synchronous stream processor type
- $runS$: the function that applies a stream processor to an input stream to produce an output stream, $S\ i\ o \to [i] \to [o]$.
- `>*<`: Parallel composition of stream processors, $S\ i_1\ o_1 \to S\ i_2\ o_2 \to S\ (i_1, i_2)\ (o_1, o_2)$

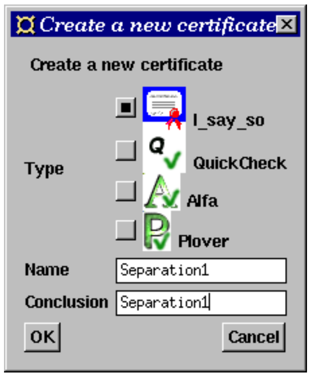
- *Separation1*: The asserted separation property, for which we want to create an Alfa certificate.

The source also includes a *certificate annotation* `{-#cert:Separation1#-}` next to the assertion, telling pfebrowser to display a certificate icon there. The icon displayed initially is a sad smiley, indicating that pfebrowser did not find a certificate of the name given in the certificate annotation.



```
assert Separation1 = ☹
  All sps1 . All sps2 . All is1 . All is2 .
  {runS (sps1>*<sps2) (zip is1 is2)} === {zip (runS sps1 is1) (runS sps2 is2)}
```

Our next step is to create an Alfa certificate by clicking on the assertion name (*Separation1*), then on the link *Create a new certificate* that appears in the status display below the source display in pfebrowser. Pfebrowser then opens a window where the type, name and conclusion of the new certificate can be entered:



Create a new certificate

Create a new certificate

Type: [✓ I_say_so] QuickCheck Alfa Plover

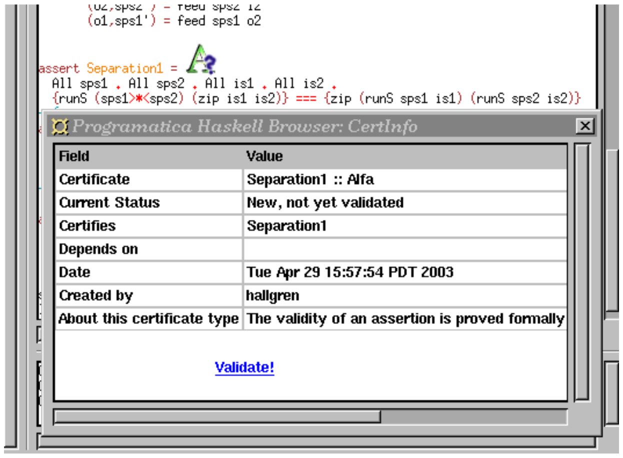
Name: Separation1

Conclusion: Separation1

We obtain a new, unvalidated certificate, and the sad smiley icon next to the assertion changed accordingly. Clicking on that icon opens the certificate info window shown in Figure 4. This window includes a link to validate the certificate. Validating the certificate might seem pointless at this point, considering that we haven't created a proof yet. It is still the best way to proceed, because it will have the effect of translating the Haskell code to Alfa and creating a template proof file where we can put our proof.

The Alfa translation is put in the subdirectory `alfa/` and the proof template file created is called `SPSProofs.alfa`. (The Alfa certificate server currently suggests one proof file per Haskell module. The user can provide alternative file names, if desired.)

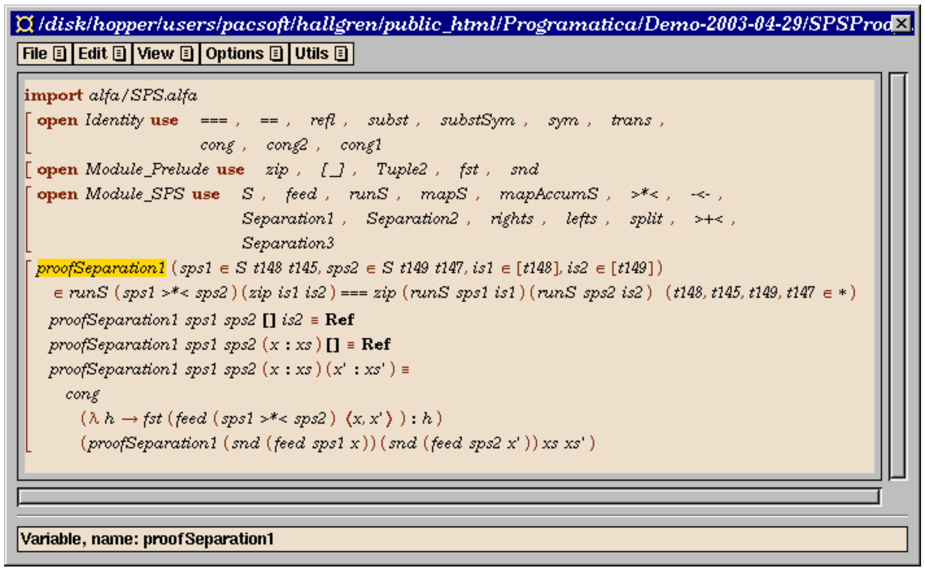
We should now proceed to edit the proof in the proof file, and eventually end up with a complete proof of the asserted property (Figure 5). We should also run



```
(o2,sps2') = feed sps2 i2
(o1,sps1') = feed sps1 o2

assert Separation1 =
  All sps1 . All sps2 . All is1 . All is2 .
  {runS (sps1>*<sps2) (zip is1 is2)} === {zip (runS sps1 is1) (runS sps2 is2)}
```

Field	Value
Certificate	Separation1 :: Alfa
Current Status	New, not yet validated
Certifies	Separation1
Depends on	
Date	Tue Apr 29 15:57:54 PDT 2003
Created by	hallgren
About this certificate type	The validity of an assertion is proved formally

Figure 4: Information window for a newly-created certificate. The icon in the background indicates a certificate for Alfa, but the question mark highlights the fact that the certificate is not yet valid.



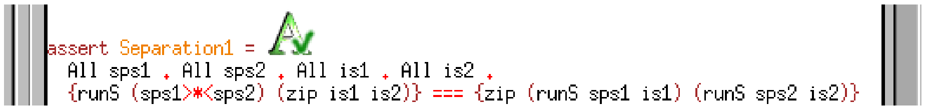
/disk/hopper/users/pacsoft/hallgren/public_html/Programatica/Demo-2003-04-29/SPSProd
File
Edit
View
Options
Utils

```
import alfa/SPS.alfa
[ open Identity use   === ,  == ,  refl ,  subst ,  substSym ,  sym ,  trans ,
                      cong ,  cong2 ,  cong1
[ open Module_Prelude use  zip ,  [_] ,  Tuple2 ,  fst ,  snd
[ open Module_SPS use  S ,  feed ,  runS ,  mapS ,  mapAccumS ,  >*< ,  -<- ,
                       Separation1 ,  Separation2 ,  rights ,  lefts ,  split ,  >+< ,
                       Separation3
[ proofSeparation1 (sps1 ∈ S t148 t145, sps2 ∈ S t149 t147, is1 ∈ [t148], is2 ∈ [t149])
    ∈ runS (sps1 >*< sps2) (zip is1 is2) === zip (runS sps1 is1) (runS sps2 is2)  (t148, t145, t149, t147 ∈ *)
   proofSeparation1 sps1 sps2 [] is2 ≡ Ref
   proofSeparation1 sps1 sps2 (x : xs) [] ≡ Ref
   proofSeparation1 sps1 sps2 (x : xs) (x' : xs') ≡
     cong
       (λ h → fst (feed (sps1 >*< sps2) ⟨x, x'⟩) : h)
       (proofSeparation1 (snd (feed sps1 x)) (snd (feed sps2 x')) xs xs')
```

Variable, name: proofSeparation1

Figure 5: Using Alfa to construct a proof of `Separation1`.

the termination checker in Alfa to verify that the proof was defined by structural induction rather than some meaningless circular construction. With a complete proof, we can rerun the validation command and find that it now succeeds and that the certificate icon is changed accordingly.



```
assert Separation1 =
  All sps1 . All sps2 . All is1 . All is2 .
  {runS (sps1>*<sps2) (zip is1 is2)} === {zip (runS sps1 is1) (runS sps2 is2)}
```

This certificate will remain valid so long as the user does not modify the definition of the property or of the functions on which it depends. If if there are any changes, however, it may be possible to revalidate the certificate automatically by rerunning the proof in `SPSProofs.alfa`.

3.2 Plover—a P-logic Verifier

Plover is an automatic verification server for Programatica. Given a Haskell program embedding one or more assertions expressed formally in *P-logic*, Plover attempts to find a proof of a designated assertion using (1) proof rules of the logic, (2) a few simple decision procedures for systems of equalities and linear inequalities, (3) a set of reduction rules for Haskell, and (4) strategies for applying these rules and decision procedures to sequent formulas in order to discharge proof obligations. Plover conducts its proof search without user interaction and it reports only its success or its failure; it does not construct a proof tree.

Plover implements *P-logic* directly. It does not depend upon a translation of Haskell into a simpler functional language nor does it translate formulas of *P-logic* into more primitive formulas. Nothing is "lost in translation."

The motivation to develop an automatic verification server lies in the boring nature of verification proofs. Unlike proofs of interesting mathematical theorems, many of which are based upon deep insights into the underlying mathematical structures, and which, once done, become the subject of continued study and improvement, verification proofs tend to be based upon shallow theories of superficially complex structures. Such proofs may not require deep insight, although they are often highly detailed. A verification proof itself is rarely the subject of ongoing study and improvement; it is the program whose properties are to be verified that is the object of primary interest. And in consequence, as a program evolves over time, the verification task is never completely finished. Proofs of a program's properties are inevitably made obsolete by its evolution and must be redone. Each of these considerations argues strongly for some degree of automation of verification proofs.

Of course, Plover, like any automated procedure that searches for proofs in a Turing-complete theory, is incomplete—its failure to find a proof does not mean that an assertion is false or even that it is unprovable. It only indicates that the strategies employed by Plover were insufficient to find a proof. In the current state of its implementation, there are many important aspects of Haskell 98 semantics for which Plover either implements no rules or lacks effective strategies. Plover does not currently support reasoning about monadic computations; it has no rules for derived operators; it lacks either decision procedures or axiomatic rules for even simple linear arithmetic; and it lacks effective strategies for verifying properties of case expressions. Rules for verifying properties of recursively-defined computations or recursive types have not yet been implemented. It does not yet comprehend data declarations.

Nevertheless, Plover is able to verify some interesting properties of programs defined using conditional expressions, tuples, non-recursive data types, function application and explicit abstraction. It comprehends (non-recursive) definitions including property definitions in *P-logic*.

3.2.1 Implementing a strategy-driven property verifier

The Plover implementation makes use of many of the techniques that have been pioneered in general-purpose theorem-provers such as ACL2, PVS, HOL and Isabelle. Internally, any theorem-prover analyzes terms representing logical formulas and rewrites them to new terms according to the rules of the logic that it implements. When viewed as a term-rewriting system, a theorem prover for an expressive logic must implement a complex, non-confluent theory. Typically, multiple choices of a next step are possible from a given term configuration. Plover is organized as a composition of programmed *strategies* for pattern-matching on terms, for term construction, and for selection among available choices.

Rules of *P-logic* are expressed in a sequent-calculus style, which supports the introduction of propositions either on the right or on the left of the turnstile in the consequent of a rule. While this style of presentation is perhaps less intuitive to the human than is natural-deduction style, with a duality between introduction and elimination rules for each form of object-language term, it allows greater flexibility in the design of an automated verifier.

Plover is implemented in Stratego, a higher-order strategy programming language. In Stratego, strategies are first-class entities, just as functions are in Haskell. This aspect of Stratego, together with a powerful library of strategies for manipulating basic term structures, for memoization, and for tasks such as capture-avoiding substitution in a lexically-scoped object term language, make it an excellent vehicle for implementing a theorem prover.

Stratego itself is implemented on a run-time library of routines for manipulating terms with maximal sharing; this gives it greater efficiency than would be gotten by building an implementation directly in a functional language such as Standard ML or Haskell.

3.2.2 Using Plover

The easiest way to invoke Plover from the Programatica tools environment is to create a certificate that names a property assertion embedded in a Programatica module, declaring the certificate to have the Plover type. This will indicate to the tool suite that Plover should be invoked when the certificate is to be validated. At that time, the Programatica tools will furnish as data the slice of the Programatica code on which the assertion depends (Section 2.4). The whole sequence of linked operations needed to validate a certificate is launched by clicking the **Validate** button in the certificate information window displayed by the pfebrowser tool.

As a caveat, one must realize that Plover (like PVS and ACL2) is an *ad hoc* theorem prover. The strategies that it employs (including its encoding of logical inference rules) are simply computer programs, and they may contain errors. Potentially, greater assurance may be obtained from a *principled* theorem prover, such as Isabelle or Coq, that interprets proof rules in a strong type theory that accepts only sound reasoning steps. However, there is a price to be paid for stringent commitment to principle. It becomes impossible to take "shortcuts" in proof construction; every proof step must be justified by evidence.

Plover relies for its claim to sound reasoning upon external proof of the soundness of rules of *P-logic*, and upon rigorous testing to provide evidence that the Plover implementation has not compromised soundness by an unfortunate programming error. (Programming errors might also compromise completeness, but since that property is never claimed, this is not a fundamental problem.)

4 Implementation Notes

Not surprisingly, our implementation has a lot in common with a Haskell compiler front-end and is likely to be reusable in many contexts outside the Programatica project. For example, it has already been used in the refactoring project at the University of Kent [25].

Our tools are implemented in Haskell, including the browser, which uses Fudgets [4] for the user interface. Amongst other things, our implementation contains an abstract syntax; a lexer and a parser; a pretty printer; some static analyses, including inter-module name resolution; some program transformations; a type checker and definition-level dependency tracking.

Some parts—the abstract syntax, the parser and the pretty printer—were inherited from another source [19] and modified, while most other parts were written from scratch. Some parts—the lexer [11] and the module system [8]—are implemented in pure Haskell 98 and can serve as reference implementations, while others make use of type system extensions, in particular multi-parameter classes with functional dependencies [14]. Together with a two-level approach to the abstract syntax [22] and other design choices, this makes the code more modular and reusable, but perhaps also too complicated to serve as a reference implementation of Haskell.

The fact that the first pass of our lexer preserves white space and comments made it easy to implement the HTML renderer. The modular structure of the lexer also allowed us to quickly create a simple tool that someone asked for on the Haskell mailing list: a program that removes comments and blank lines from Haskell code [12].

While implemented from scratch, key design choices in the type checker were influenced by the simplicity of *Typing Haskell in Haskell* [15], the efficiency of the type checkers in HBC and NHC, and the constraint based approach used in one of Johan Nordlander's type checkers for O'Haskell [20]. It performs the dictionary translation and inserts enough type annotations to make the output suitable for translation to an explicitly typed language like Structured Type Theory or System F.

5 Further Information and Availability

More information on the Programatica Project is available from our web pages [21]. Preliminary versions of the tools and user documentation can be downloaded from [23].

References

[1] Andreas Abel. foetus – Termination Checker for Simple Functional Programs. `www.tcs.informatik.uni-muenchen.de/~abel/foetus/`, 1998. Programming Lab Report.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] Ana Bove. *General Recursion in Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Gteborg, Sweden, 2002. `www.cs.chalmers.se/~bove/Papers/phd_thesis.ps.gz`.

[4] Magnus Carlsson and Thomas Hallgren. Fudgets. `www.cs.chalmers.se/Fudgets/`, 1993-2003.

[5] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000. `citeseer.nj.nec.com/claessen99quickcheck.html`.

[6] Thierry Coquand. Structured type theory. `www.cs.chalmers.se/~coquand/STT.ps.Z`, June 1999. Preliminary version.

[7] Maarten de Mol. Sparkle. `www.cs.kun.nl/Sparkle/`, 2003.

[8] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification for the Haskell 98 Module System. In *Proceedings of the 2002 Haskell Workshop*, Pittsburgh, USA, October 2002. `www.cse.ogi.edu/~diatchki/hsmod/`.

[9] The Glasgow Haskell Compiler, 2004. `www.haskell.org/ghc/`.

[10] Thomas Hallgren. Home Page of the Proof Editor Alfa. `//www.cs.chalmers.se/~hallgren/Alfa/`, 1996-2003.

[11] Thomas Hallgren. A Lexer for Haskell in Haskell. `www.cse.ogi.edu/~hallgren/Talks/LHiH`, 2002.

[12] Thomas Hallgren. stripcomments. `www.cse.ogi.edu/~hallgren/stripcomments/`, 2002.

[13] Hugs Online. `www.haskell.org/hugs/`, 2004.

[14] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, number 1782 in LNCS, Berlin, Germany, March 2000. Springer-Verlag.

[15] M.P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Paris, France, September 1999. `www.cse.ogi.edu/~mpj/thih/`.

[16] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, April 2003. ISBN 0521826144, `www.haskell.org/definition/`.

[17] Richard B. Kieburtz. P-logic: property verification for Haskell programs. `//www.cse.ogi.edu/PacSoft/projects/programatica/papers.htm`, February 2002.

[18] Simon Marlow. Haddock. `www.haskell.org/haddock/`, 2003.

[19] Simon Marlow et al. The hssource library. Distributed with GHC [9].

[20] Johan Nordlander. O'haskell. `www.cs.chalmers.se/~nordland/ohaskell/`, 2001.

[21] The Programatica Project home page. `www.cse.ogi.edu/PacSoft/projects/programatica/`, 2002.

[22] Tim Sheard. Generic unification via two-level types and parameterized modules. In *International Conference on Functional Programming*, pages 86–97, 2001. `citeseer.nj.nec.com/451401.html`.

[23] The Programatica Team. Programatica Tools. `www.cse.ogi.edu/~hallgren/Programatica/download/`, August 2003.

[24] The Programatica Team. Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell. In *Proceedings of the High Confidence Software and Systems Conference*. National Security Agency, April 2003. Available via [21].

[25] Simon Thompson, Claus Reinke, et al. Refactoring Functional Programs. `www.cs.kent.ac.uk/projects/refactor-fp/`, 2003.