Implementation of Proof Search in the Imperative Programming Language Pizza

Christian Urban

Computer Laboratory, University of Cambridge,
Gonville and Caius College, Cambridge CB2 1TA, UK.
Christian.Urban@cl.cam.ac.uk

Abstract. Automated proof search can be easily implemented in logic programming languages. We demonstrate the technique of success continuations, which provides an equally simple method for encoding proof search in imperative programming languages. This technique is exemplified by developing an interpreter for the calculus G4ip in the language Pizza.
Keywords: Success Continuations, G4ip, Pizza

1 Introduction

A sequent-style formulation of a logic calculus is a convenient starting-point for automating proof search because the corresponding inference rules are 'local' operations on proofs. A sequent can be proved by applying inference rules until one reaches axioms, or can make no further progress in which case one must backtrack or even abandon the search. This proving method is a simple depth-first strategy; it is preferred over a less efficient breadth-first strategy. However, this method requires the mechanism of choice points in order to facilitate the backtracking. Logic programming languages provide substantial support for depth-first proof search and therefore simplify considerably the implementation of a logical calculus. Unfortunately these languages have a rather limited support for user interfaces, which makes them unsuitable for larger applications. Imperative programming languages, on the other hand, provide a rich environment for user interfaces, but seem to need a significant overhead of code for an implementation of a logic calculus. We illustrate in the paper an implementation technique using success continuations which provides a simple implementation of a logic calculus. The technique is not new: it was introduced in [Car84] with a rather technical illustration in LISP. Later an excellent paper [EP91] appeared which implements a full-fledged interpreter for λProlog in SML.

In the paper we focus on intuitionistic logic for which Gentzen's calculus LJ (also G1i) is a standard formalisation. Unfortunately an interpreter for LJ using a depth-first search strategy cannot guarantee termination. Several modifications can be made to the inference rules of LJ without loss of soundness and completeness. As result an efficient depth-first proof search can be designed for the propositional fragment of intuitionistic logic. The name G4ip has been assigned to the corresponding calculus in [TS96]. This calculus is also known as LJT which has been studied thoroughly in [Dyc92]. The inference rules of G4ip are given in Figure 1. G4ip has the pleasant property that a depth-first proof search does not loop and that it is sound and complete in terms of provable theorems (wrt. G1ip).

In the paper we demonstrate the implementation technique using the object oriented programming language Pizza. Pizza is an extension of Java with some new features. Although Pizza is a superset of Java, Pizza programs can be translated into Java code or can be compiled into ordinary Java Byte Code (see [OW97] for a technical introduction into Pizza). We make use of the following two new language constructs:

- higher-order functions, i.e. functions may be passed as parameters or returned from methods;
- class cases and pattern matching, which allow much simpler and readable code.

$$\frac{}{A,\Gamma\vdash A}\;\mathit{Axiom} \qquad \frac{\Gamma\vdash B \quad \Gamma\vdash C}{\Gamma\vdash B\&C}\;\&\text{-R} \qquad \frac{\Gamma\vdash B_i}{\Gamma\vdash B_1\vee B_2}\;\vee\text{-R}_i \qquad \frac{B,\Gamma\vdash C}{\Gamma\vdash B\supset C}\;\supset\text{-R}$$

$(A$ being atomic$)$

$$\frac{}{\mathit{false},\Gamma\vdash G}\;\mathit{false}\text{-L} \qquad \frac{B,C,\Gamma\vdash G}{B\&C,\Gamma\vdash G}\;\&\text{-L} \qquad \frac{B,\Gamma\vdash G \quad C,\Gamma\vdash G}{B\vee C,\Gamma\vdash G}\;\vee\text{-L} \qquad \frac{B,A,\Gamma\vdash G}{A\supset B,A,\Gamma\vdash G}\;\supset\text{-L}_1$$

$(A$ being atomic$)$

$$\frac{C\supset(D\supset B),\Gamma\vdash G}{(C\&D)\supset B,\Gamma\vdash G}\;\supset\text{-L}_2 \qquad \frac{C\supset B,D\supset B,\Gamma\vdash G}{(C\vee D)\supset B,\Gamma\vdash G}\;\supset\text{-L}_3 \qquad \frac{D\supset B,\Gamma\vdash C\supset D \quad B,\Gamma\vdash G}{(C\supset D)\supset B,\Gamma\vdash G}\;\supset\text{-L}_4$$

Fig. 1. The inference rules of G4ip.

These features are not directly present in Java, but Pizza makes them accessible by translating them into Java. The higher-order functions are essential for our implementation.The success continuations are functions passed as parameters or returned as values. Pizza provides the programmer with access to the same extensive libraries for graphic and network applications as Java. The paper assumes some familiarity with Java or Pizza.

2 Design of the Proving Method

In this section we are concerned with the abstract design of the proof search. Each step in the depth-first search is a reversed application of an inference rule. The inference rule analyses one formula, called the principal formula, of the sequent being proved. The construction of proofs, however, is not deterministic; it does not determine the choice of the principal formula. All formulae in a sequent are possible candidates and a naive proving method has to explore them all. A widely used optimisation is the search for uniform proofs (see [MNPS91]). Uniform proofs are constrained on the choice of the principal formula, i.e.,whenever sequents have non-atomic goal formulae then only inference rules are chosen which analyse these goal formulae. This optimised search strategy can be employed to logics for which uniform proofs are complete. However for G4ip, this optimisation cannot be made. For example, the sequent $p\vee q\vdash p\vee q$ has only one proof which is non-uniform. Therefore, we shall implement a proving method which first enumerates all formulae from the left-hand side of the sequent as being principal and applies corresponding left rules. Subsequently the goal formula is chosen and a right rule is applied. The sequents of G4ip are specified with multisets of formulae on the left-hand side. Accordingly, the enumeration of formulae as being principal can be done in any order. We have avoided making some optimisations in favour of a clear illustration of the success continuation technique. A λProlog implementation of the outlined proving method has been provided in order to compare the technique of success continuations and more traditional implementations using logic programming (see Section 5). A similar implementation was presented in [HM94] using Lolli, a logic programming language based on linear logic. This implementation differs from ours in the inactive parts of the sequents (i.e. Γ) which are treated implicitly.

3 The Representation of Formulae and Sequents

Amongst the new language features of Pizza are class cases and pattern matching, which provide a very pleasant syntax for algebraic data types. The formulae of G4ip are specified by the following grammar:

$$F ::= \quad \mathit{false} \quad | \quad A \quad | \quad F\&F \quad | \quad F\vee F \quad | \quad F\supset F$$

The class cases allow a straightforward implementation of this specification; it is analogous to the SML implementation of λProlog's formulae in [EP91]. The class of formulae is given below. On the right-hand side two examples illustrate the use of this class:

```
public class Form {
   case False();
   case Atm(String c);
   case And(Form c1,Form c2);
   case Or(Form c1,Form c2);
   case Imp(Form c1,Form c2);
}
```

$p \supset p$ is represented as:
`Imp(Atm("p"),Atm("p"))`

$a \vee (a \supset \mathit{false})$ is represented as:
`Or(Atm("a"),Imp("a",False()))`

The class cases of Pizza also support an implementation of formulae specified by a mutually recursive grammar. This is required, for example, when implementing hereditary Harrop formulae.

The sequents of G4ip, which have the form $\Gamma \vdash G$, are represented by means of the class below. The left-hand sides are specified by multisets of formulae. Therefore, we do not need to worry about the order in which the formulae occur.

```
public class Sequent {
   Form    G;
   Context Gamma;

   public Sequent(Context _Gamma, Form _G) {...}
}
```

We have a constructor for generating new sequents during proof search. `Context`[1] is a class which represents multisets. This class provides methods for adding elements to a multiset (`add`), taking out elements from a multiset (`removeElement`) and testing the membership of an element in a multiset (`includes`).

4 The Implementation of the Proof Search

In a first attempt we could implement the choice of a principal formula and the application of an inference rule in a recursive style. Consider the following two pictures:

$$\frac{\vdots}{F_1,\ldots,F_{n-1} \vdash F_n}\,R_i \qquad\qquad \frac{\overset{\vdots}{\mathit{Premise}_1}\ldots\overset{\vdots}{\mathit{Premise}_n}}{\mathit{Conclusion}}\,R$$

and suppose we have a function *prove*. This function receives a sequent as the only argument. In prove, we could enumerate all formulae of the sequent (F_1 to F_n) as being principal and apply a corresponding inference rule. This produces some new sequents ($\mathit{Premise}_1$ to $\mathit{Premise}_n$) to be proved. Therefore we perform prove again by calling it recursively with $\mathit{Premise}_1$ to $\mathit{Premise}_n$ as arguments. However this simple method intermingles the two concepts of proof obligations (which must be proved) and choice points (which can be tried out to construct a proof). In order to make this first attempt work we need to make some non-trivial modifications. A simpler method is to add another argument to the function prove. This additional argument will be an anonymous function, which is permitted by Pizza and can be locally defined. The function prove is now of the form *prove(sequent,sc)*. Somewhat simplified the first argument is the leftmost premise ($\mathit{Premise}_1$) and the second argument *sc*, the *success continuation*, represents the other proof obligations ($\mathit{Premise}_2$ to $\mathit{Premise}_n$). In case we succeed in proving the first premise we then can attempt to prove the other premises. For the success continuations we use higher-order functions. This allows us to impose a certain structure (or order) on the success continuations.
The inference rules fall into three groups: inference rules with a single premise, inference rules with two premises and inference rules without premises (e.g. *Axiom*). Suppose we have called *prove* with a sequent *s* and a success continuation *sc*. The inference rules of the first group manipulate *s*

---

[1] The class Context is a simple extension of the class Vector available in the Java libraries.

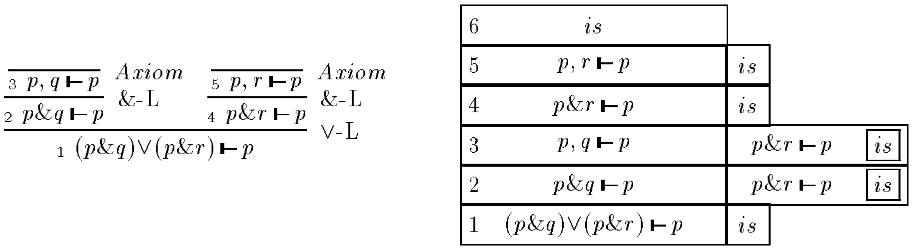

Fig. 2. An Example for the Proving Method.

obtaining s' and call *prove* again with the new sequent s' and the current success continuation *sc* (Steps 2-3 and 4-5 in Figure 2). The inference rules of the second group have two premises, s_1 and s_2. These rules call *prove* with s_1 and the new success continuation *prove*(s_2,*sc*) (Step 1-2 in Figure 2). The third group of inference rules only invoke the success continuation if the rule is applicable (Steps 3-4 and 5-6 in Figure 2).

We are going to give a detailed description of the code for the rules: &-L, $\vee$-R$_i$, $\vee$-L and *Axiom*. For lack of space the code for the other rules is omitted. The function *prove* enumerates all formulae as being principal and two switch statements select a corresponding case depending on the form and the occurrence of the principal formula. The &-L rule is in the first group; it modifies the sequent being proved and calls *prove* again with the current success continuation *sc*. The code is as follows:[2]

```
case And(Form B, Form C):    //&-L
  prove(new Sequent(Gamma.add(B,C),G),sc); break;
```

The $\vee$-R$_i$ rule is an exception in the first group. It breaks up a goal formula of the form $B_1 \vee B_2$ and proceeds with one of its component. Since we do not know in advance which component leads to a successful proof we have to try both. Therefore this rule acts as a choice point, which is encoded by a recursive call of *prove* for each case.

```
case Or(Form B1,Form B2):    //∨-Ri
  prove(new Sequent(Gamma,B1),sc);
  prove(new Sequent(Gamma,B2),sc); break;
```

The $\vee$-L rule falls into the second group where the current success continuation, *sc*, is modified. It calls *prove* with the first premise $(B, \Gamma \vdash G)$ and wraps up the success continuation with the new proof obligation $(C, \Gamma \vdash G)$. The construction `fun()->void {...}` defines an anonymous function: the new success continuation. In case the sequent $B, \Gamma \vdash G$ can be proved, this function is invoked.

```
case Or(Form B,Form C):    //∨-L
  prove(new Sequent(Gamma.add(B),G),
        fun()->void {prove(new Sequent(Gamma.add(C),G),sc);} ); break
```

The *Axiom* rule falls into the third group. It first checks if the principal formula (which is an atom) matches with the goal formula and then invokes the success continuation *sc* in order to prove all remaining proof obligations.

```
case Atm(String c):    //Axiom
  if (G instanceof Atm) { if (G.c.compareTo(c) == 0) { sc(); }
  } break;
```

---

[2] `Gamma` stands for the set formulae on the left-hand side of a sequent excluding the principal formula; `G` stands for the goal formula of a sequent.

The proof search is started with an initial success continuation *is* (cf. 2). This initial success continuation is invoked when a proof has been found. In this case we want to give some response to the user, an example for the initial success continuation, *is*, could be as follows:

```
public void initial_sc() { System.out.println("Provable!"); }
```

Suppose we attempt to start the proof search with $prove(p, p \vdash p, is)$. We would find that the prover responds twice with `"Provable!"`, because it finds two proofs. In our implementation this problem is avoided by encoding the proof search as a thread. Whenever a proof is found, the initial success continuation displays the proof and suspends the thread. The user can decide to resume with the proof search or abandon the search.

5 Conclusion

We have adapted the technique of success continuations presented in [Car84] and [EP91] and provided an implementation of G4ip in the imperative programming language Pizza. The implementation cannot be considered as optimal in terms of speed. A much more efficient algorithm for G4ip (but less clear) has been implemented by Dyckhoff in Prolog. Similar ideas can be encoded in our Pizza implementation; but our point was not the efficiency but the clarity of the implementation using success continuations. The technique is applicable elsewhere whenever backtracking is required. We compared the code of our implementation with an implementation in λProlog: the ratio of code is approximately 2 to 1. This result is partly due to the fact that we had to implement a class for multisets. In a future version of Java, we could have accessed a package in the library. The technique of success continuation can also be applied to a first-order calculus as shown in [EP91], but the required mechanism of substitution needs to be implemented separately.

The code of the implementations and some accompanying information are available under the address `http://www.cl.cam.ac.uk/~cu200/Prover/`. It includes an applet which can be executed on a Java-capable browser. Some problems arose from the different level of Java support in different browsers. We had to make some compromises in order to support as many platforms as possible. This should change with the release of new browsers and a stable Java-specification (resp. Pizza-specification).

Acknowledgements: I am very grateful for Dr Roy Dyckhoff's constant encouragement and many comments on my work. I thank Dr Gavin Bierman who helped me to test the prover applet. The work was supported by a scholarship from the German Academic Exchange Service.

References

[Car84] Mats Carlsson. On Implementing Prolog in Functional Programming. *New Generation Computing*, pages 347–359, 1984.

[Dyc92] R. Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *Journal of Symbolic Logic*, 57(3):795–807, September 1992.

[EP91] Conal Elliott and Frank Pfenning. A Semi-Functional Implementation of a Higher-Order Logic Programming Language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–352. MIT Press, 1991.

[HM94] Joshua Hodas and Dale Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure Applied Logic*, 51:125–157, 1991.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[TS96] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996.