Generating Quantifiers and Negation to Explain Homework Testing

Jason Perry and **Chung-chieh Shan**
Rutgers University
Department of Computer Science

Abstract

We describe Prograder, a software package for automatic checking of requirements for programming homework assignments. Prograder lets instructors specify requirements in natural language as well as explains grading results to students in natural language. It does so using a grammar that generates as well as parses to translate between a small fragment of English and a first-order logical specification language that can be executed directly in Python. This execution embodies multiple semantics—both to check the requirement and to search for evidence that proves or disproves the requirement. Such a checker needs to interpret and generate sentences containing quantifiers and negation. To handle quantifier and negation scope, we systematically simulate continuation grammars using record structures in the Grammatical Framework.

1 Introduction

The typical programming assignment in a computer-science course comes with not only a problem description but also correctness and stylistic requirements such as

(1) Every source file compiles and has comments, and a text file mentions every source file and every header file.

Although the requirements do not usually specify exactly how the students' work will be judged, they make it much easier to grade and respond to the submitted work. Many requirements can be checked automatically by computer, and often they are—perhaps even right after each student uploads their work so that the student can revise their work using the immediate feedback.

This common workflow is wanting in two aspects. First, the requirements are both specified to the students in English and coded into a testing harness by the course staff. Keeping the two versions is a hassle that involves much boilerplate text as well as boilerplate code. Second, students rightfully demand comprehensible explanations when their work is rejected by the requirement tester. It is tricky to code up a tester that produces error messages neither too terse nor too verbose, when one student might forget to comment just one file and another might not know the lexical syntax of comments at all.

A natural approach to improve this workflow, then, is to specify the requirements in a formal language and to implement an interpreter for the language that produces explanations. Because many instructors, like students, are averse to learning new languages, we pursue the use of a controlled subset of English as that formal language. In short, we aim to produce a programming-assignment tester that allows instructors to specify requirements for programming assignments in natural language, checks those requirements automatically by executing commands on the submitted assignment files, and generates for the student a natural-language explanation of the results. For example, in an introductory programming class, a professor may write the specification sentence (1). The system would then grade all students' programming assignments according to these criteria, and return individualized explanations

Proceedings of the NAACL HLT 2010 Fifth Workshop on Innovative Use of NLP for Building Educational Applications, pages 57–65,
Los Angeles, California, June 2010. ©2010 Association for Computational Linguistics

to students like

(2) Credit was lost because `bar.c` did not compile and no text file mentioned the files `foo.h` and `baz.h`.

As this example illustrates, the tester needs to interpret and generate sentences with quantifiers and negation. Although the interpretation of quantifiers and negation is a traditional research area in computational linguistics (VanLehn, 1978; Hobbs and Shieber, 1987; Moran, 1988), their generation is much less studied (Gailly, 1988). Even if our system were to compose explanations entirely from the input specification sentences and their negation, it cannot negate a specification sentence merely by adding or removing verbal auxiliaries: the negation of "a source file defines `main()`" is not "a source file does not define `main()`".

1.1 Contributions

We have built Prograder, a rudimentary programming-assignment tester that correctly interprets and generates a small fragment of English that includes quantifiers and negation. This paper describes the architecture of Prograder. Our implementation uses a declarative grammar that simultaneously supports interpretation and generation by relating English phrase structure to type-logical semantics. This paper details the new techniques we use in this grammar to represent quantifier scope and De Morgan duality in a tractable way.

Prograder also lets us investigate how to make a computer program explain its own execution. The concerns for a tester that must justify its output to students are not merely grammatical, but involve the semantics and pragmatics of summarization and justification. For example, when is it semantically correct to combine entities within an NP, as in "`foo.h` and `baz.h`" above? When is an explanation pragmatically enough for a student who has a right to know why she lost credit on a programming assignment? Which pieces of evidence must be stated and which are better left out? We plan to study these questions within type-logical semantics as well.

1.2 Organization of This Paper

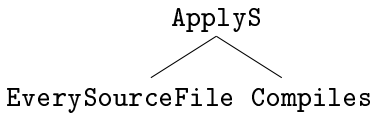
In Section 1, we have introduced the problem and outlined the contributions of the research carried out in building Prograder. In Section 2, we give a high-level overview of the architecture of the system, including the representation of natural-language syntax and type-logical semantics, as well as the procedure of operation. Section 3 describes Prograder's checking procedure and how it generates explanations in tandem with the checking. In Section 4, we begin to describe the details of the grammatical framework that allows us to handle bidirectional translation between English and the logical form. Continuing in Section 5, we discuss the handling of negation and quantifier scoping by means of a record-based implementation of continuation grammars. Section 6 reviews the architecture of the system with additional implementation details, and Section 7 concludes with future work.

2 System Overview

We explain the architecture of our Prograder system using a simplified example. Suppose that the instructor specifies the requirement that

(3) Every source file compiles.

Prograder begins by parsing this sentence into an abstract syntax tree:

(4)


Interpreting this tree compositionally gives the meaning of the sentence:

(5)
```
everysourcefile(lambda x:
                  compiles(x))
```

On one hand, this expression is a formula in predicate logic. In general, each requirement specification is a sentence, whose truth value is determined by checking a single student's programming assignment. We use the types of Montague grammar (1974), which are the base type of entities e, the base type of propositions t, and function types notated by $\rightarrow$. Our logical language is defined by a domain-specific vocabulary of typed predicates and entities (Hudak, 1996). Naturally, the files submitted by the student are entities.

For example, the unary predicates `compiles` and `hascomments` have the type $e \to t$, and the binary predicate `mentions` has the type $e \to e \to t$. These predicates in our vocabulary represent various properties of submitted files that instructors may want to check. Logical connectives are also part of the vocabulary; for instance, `and` and `or` have the type $t \to t \to t$, `not_b` has the type $t \to t$, and `everysourcefile` has the type $(e \to t) \to t$. Therefore, the expression (5) has the type t, as do the expressions

(6)
```
compiles("foo.c")
```

(7)
```
and(compiles("foo.c"),
    hascomments("bar.c"))
```

On the other hand, the expressions (5–7) are executable Python code. Prograder includes a library of Python functions such as `everysourcefile`, which iterates over the source files submitted, and `compiles`, which invokes `gcc`. As each student submits their homework, we evaluate the expression (5) to check the requirement (3). We use Python's own lambda abstraction and first-class functions to express quantified statements. For example, evaluating (5) invokes `gcc` on each source file submitted.

Evaluation not only computes a Boolean value but also yields a tree of evidence statements to justify the result. At the root of the tree is either the requirement (5) or its negation:

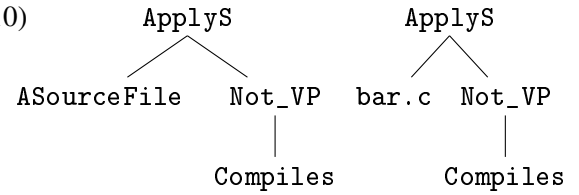
(8)
```
asourcefile(lambda x:
              not_b(compiles(x)))
```

Each node's children are the premises that together justify the conclusion at that node. For example, if every source file submitted by the student compiles successfully except one, then (8) would be justified by a sole child:

(9)
```
not_b(compiles("bar.c"))
```

Prograder then parses each evidence statement into abstract syntax:

(10)



From these trees, Prograder finally renders each evidence statement as an English clause:

(11) A source file does not compile because `bar.c` does not compile.

To summarize, Prograder's steps of operation are as follows:

1. Parse the natural-language requirement statement into a phrase-structure syntax tree.
2. Produce a logical semantic representation from the syntax tree.
3. Execute the semantic representation to check the requirement and produce a semantic representation of each evidence statement.
4. Parse the evidence statements into phrase-structure syntax trees.
5. Produce natural-language explanation from the syntax trees.

From end to end, these steps operate on just three levels of representation: the abstract syntax trees in (4) and (10) can be spelled out either in natural language (English) as in (3) and (11) or in predicate logic (Python) as in (5–9). Thus, Prograder interprets natural-language requirements and generates natural-language explanations using the same executable semantic representations. In fact, it uses the same grammar, as described in Section 4.

3 Gathering Evidence while Testing Truth

As sketched above, evaluating a proposition gives a Boolean result along with a tree of evidence propositions that justify that result. That is, we represent the type t in Python as an ordered pair, not just a Boolean value. This is straightforward for primitive first-order predicates. For example, the `compiles` function, when invoked with the argument `"foo.c"`, tries to compile `foo.c`, then either returns `True` along with a trivial evidence tree containing just the proposition `compiles("foo.c")`, or returns `False` along with a trivial evidence tree containing just the proposition `not_b(compiles("foo.c"))`. In short, we hold whether a file compiles to be self-evident.

The picture is more complex for logical connectives, especially higher-order functions such as the quantifier `everysourcefile`. Ideally, the falsity of (3) should be justified by an explanation like

(12) Not every source file compiles because `foo.c` and `bar.c` do not compile.

Also, the explanation should be generated by composing the implementations of `everysourcefile` and of `compiles`, so that new quantifiers ("most source files") and predicates ("has comments") can be added as separate modules. For example, evaluating the proposition `not_b(compiles("foo.c"))` first evaluates `compiles("foo.c")` then negates the Boolean while keeping the evidence the same.

Prograder currently justifies quantified propositions in the following compositional way. When the higher-order function `everysourcefile` is invoked with the predicate argument `p`, it invokes `p` on each source file submitted and collects the resulting Boolean-evidence pairs. If any of the Boolean results is `False`, then the overall Boolean result is of course also `False`. This result is justified by an evidence tree whose root proposition is

(13) `not_b(everysourcefile(p))`

and whose subtrees are the evidence trees for all the `False` results. (After all, the primary mode of explanation required in grading a programming assignment is to say why credit has been subtracted.) Prograder then produces the serviceable explanation

(14) Not every source file compiles because `foo.c` does not compile and `bar.c` does not compile.

(We are working on simplifying this explanation to (12).) This process generalizes immediately to propositions with multiple quantifiers, such as

(15) Every text file mentions a source file.

```
everytextfile(lambda x:
  asourcefile(lambda y:
    mentions(y)(x)))
```

For example, Prograder might explain that

(16) Not every text file mentions a source file because `README` mentions no source file.

In sum, Prograder generates explanations in the same process—the same sequence of function calls and returns—as it computes Boolean values. In this way, the checking process gains an additional interpretation as a process of gathering evidence for the explanation. The explanation itself is a tree structure corresponding to the pattern of function calls in the computation; in fact, it is a proof tree for the requirements specification statement (or its negation). Once all the evidence has been gathered, a textual version of the explanation can be generated by traversing the tree and concatenating evidence statements, possibly using summarization techniques to make the explanation more concise.

We have so far glossed over how English sentences, especially those with quantification and negation such as (14) and (16), are parsed into and generated from logical representations. The rest of this paper describes our principled approach to this problem, based on a consistent mapping between syntactic categories and semantic types.

4 Using the Grammatical Framework

We relate natural language (English) to logical semantics (Python) using the Grammatical Framework (GF) of Ranta (2004). GF is a mature system that allows linguists and logicians to define grammars for both natural and formal languages using a Haskell-like syntax. Once defined, the grammars can be used for parsing as well as generation (*linearization*) with no further programming.

In GF, grammars are specified in two parts: an abstract grammar and a concrete grammar. An abstract grammar specifies the set of well-formed abstract syntax trees, whereas a concrete grammar specifies how to spell out abstract syntax as strings. For example, an abstract grammar can admit the abstract syntax tree in (4) by specifying that `EverySourceFile` is an NP, `Compiles` is a VP, and `ApplyS` combines an NP and a VP into an S:

```
fun EverySourceFile: NP;
fun Compiles: VP;
fun ApplyS: NP -> VP -> S;
```

A concrete grammar for English can then specify that the linearization of `EverySourceFile` is a singular (`Sg`) string,

```
lin EverySourceFile = {
  s = "every source file"; n = Sg };
```

the linearization of `Compiles` is a pair of strings,

```
lin Compiles = {
  s = { Sg => "compiles";
        Pl => "compile" } };
```

and the linearization of `ApplyS` is a function that combines these two linearizations into the string (3).

```
lin ApplyS NP VP = {
  s = NP.s ++ (VP.s ! NP.n) };
```

Here `++` denotes string concatenation and `!` denotes table lookup.

Multiple concrete grammars can share the same abstract grammar in GF, as in synchronous grammars (Aho and Ullman, 1969) and abstract categorial grammars (de Groote, 2002). This sharing is meant to enable multilingual applications, in which the same meaning representation defined by a single abstract grammar can be rendered into various natural languages by different concrete grammars.

This separation into abstract and concrete grammars lets us use one concrete grammar to model English and another to model the Python syntax of Prograder's logical forms. For example, the linearization of `Compiles` into Python is simply `compiles`, which can be combined with `"foo.c"` by the linearization of `ApplyS` to form `compiles("foo.c")`. The system can thus parse a natural-language specification into an abstract syntax tree using the first concrete grammar, then produce the corresponding semantic representation using the second concrete grammar. Since each concrete grammar can be used for both parsing and generation, the system can also be run in the other direction, to parse the semantic representations of an explanation, then generate that explanation in natural language.

Since the linearization `compiles("foo.c")` in Python is virtually isomorphic to its abstract syntax tree, one may wonder why we bother with the second concrete grammar at all. Why not just express the type-theoretical semantic structure in the abstract syntax and relate it to English in a single concrete grammar? The answer is that using two concrete grammars lets us represent quantifier meanings and scope preferences. Without the distinction between abstract syntax and logical semantics, we would have to specify how to linearize `everysourcefile` and `asourcefile` so that the Python in (15) linearizes to the English. Moreover, Prograder should disprefer the inverse-scope reading

(17)
```
asourcefile(lambda y:
      everytextfile(lambda x:
        mentions(y)(x)))
```

for the same English sentence. We do not see a way to achieve these goals given GF's limited support for higher-order abstract syntax. Instead, we keep our grammars first-order and let our abstract grammar express only the surface structure of English, where quantifiers stay "in situ" just like proper names.

Below we describe how even a first-order concrete grammar for semantic representations can represent quantifier meanings and scope preferences.

5 Quantifier Scope, Negation and Continuation Grammars

Quantifier scoping has long been a key source of difficulty in mapping natural language sentences to logical forms. Scope ambiguities arise even in relatively simple sentences such as (15), which any instructor might be expected to generate in specifying programming assignment requirements. The scope of negation is also problematic. An algorithm for generating all possible quantifier scopings was detailed by Hobbs and Shieber (1987). However, we need a solution that prefers one highly likely default scoping, that supports both interpretation and generation, and that is integrated with the type structure of our semantic representation.

Compositional semantics based on *continuations* (Barker, 2002) can represent preferred scoping of quantifiers and negation without the semantic type-shifting or syntactic underspecification (Hendriks, 1993; Steedman, 1996; Bos, 1995; Koller et al., 2003) that typically complicates interpreting and generating quantification. The rough idea is to generalize Montague's PTQ (1974), so that every constituent's semantic type has the form $(\cdots \to t) \to t$: not only does every NP denote the type $(e \to t) \to t$ instead of e, but every VP also denotes the type $((e \to t) \to t) \to t$ instead of $e \to t$. For example (as in PTQ),

(18) "foo.c" denotes `lambda c: c("foo.c")`

(19) "every text file" denotes
```
lambda c:
  everytextfile(lambda x: c(x))
```

Analogously (but unlike in PTQ),

(20) "compiles" denotes `lambda c: c(compiles)`

(21) "mentions a source file" denotes

```
lambda c:
  asourcefile(lambda x:
    c(mentions(x)))
```

Recall from Section 4 that we model denotations as the linearizations of abstract syntax trees. Therefore, in GF, we want to specify linearizations like

```
lin EverySourceFile =
  lambda c:
    everysourcefile(lambda x: c(x));
lin Compiles = lambda c: c(compiles);
```

to be combined by the linearization of `ApplyS`

```
lin ApplyS NP VP =
  NP(lambda n: VP(lambda v: v(n)));
```

into the expression (5).

In this last linearization, the innermost application `v(n)` means to apply the VP's predicate meaning to the subject NP's entity meaning. The surrounding `NP(lambda n: VP(lambda v: ...))` lets a quantificational subject take wide scope over the VP. This general composition rule thus yields surface scope to the exclusion of inverse scope. In particular, it equally well combines the denotations in (19) and (21) into the expression (15), rather than (17). In the present implementation of Prograder, the rules all encode surface scope for the quantifiers. A similar linearization forms the VP denotation in (21) by composing a transitive verb and its object NP:

```
lin ApplyVP VT NP =
  lambda c: NP(lambda n: c(VT(n)))
```

The same machinery generalizes to handle possessives, ditransitive verbs, relative clauses, and so on.

5.1 Simulating higher-order functions

As shown above, denotations using continuations are higher-order functions. However, linearization in GF does not allow higher-order functions—in fact, the only "functions" allowed in GF are record or table structures indexed by a finite set of parameters. To keep parsing tractable, GF only lets strings be concatenated, not beta-reduced as lambda-terms; the one extension that GF makes to the context-free model is allowing argument suppression and repetition. In other words, GF cannot equate logical forms by beta-equivalence. Therefore, we cannot just feed the pseudocode above into GF to generate explanations. This is an instance of the problem of logical-form equivalence (Shieber, 1993).

Fortunately, because denotations using continuations are always of a certain form (Danvy and Filinski, 1992), we can simulate these higher-order functions using first-order records. Specifically, we simulate a higher-order function of the form

(22) `lambda c:` s_l `c(`s_m`)` s_r

by the *triple of strings*

(23) `{ s_l =` s_l`; s_m =` s_m`; s_r =` s_r `}`

in GF. The middle string s_m corresponds to the "core" of the phrase, and the left and right strings s_l, s_r are those parts which may take scope over other phrases. For example, following the pseudocode above, we write

```
lin EverySourceFile = {
  s_l = "everysourcefile ( lambda x :";
  s_m = "x";
  s_r = ")" };
lin Compiles = {
  s_l = "";
  s_m = "compiles";
  s_r = "" };
```

in our GF concrete grammar. Continuing to follow the pseudocode above, we can also implement the linearizations of `ApplyS` and `ApplyVP` to operate on triples rather than the functions they simulate:

```
lin ApplyS NP VP = {
  s = NP.s_l ++ VP.s_l ++
      VP.s_m ++ "(" ++ NP.s_m ++ ")" ++
      VP.s_r ++ NP.s_r };
lin ApplyVP VT NP = {
  s_l = NP.s_l;
  s_m = VT.s ++ NP.s_m;
  s_r = NP.s_r };
```

Here the linearization of the transitive verb `VT` consists of a single string `s`.

This simulation is reminiscent of Barker and Shan's "tower notation" (2008). In general, we can simulate a n-level semantic tower by a tuple of

$2n-1$ strings. Overall, this simulation makes us hopeful that linearization in GF can be extended to a broad, useful class of higher-order expressions while keeping parsing tractable.

5.2 Maintaining De Morgan duals

Both to interpret requirements and to generate explanations, our system needs to deal with negation correctly. Whether in the form of a negative quantifier such as "no source file" or a VP modifier such as "don't", negation takes scope. (In the case of the determiner "no", the scope of negation is linked to the containing NP.) We use continuations to account for the scope of negation, as for all scope-taking.

When scope ambiguities arise, negation exhibits the same preference for surface scope as other quantifiers. For example, all of the sentences below prefer the reading where the subject takes wide scope.

(24) A source file doesn't compile.

(25) A text file mentions no source file.

(26) No text file mentions a source file.

The linearization of `ApplyS` shown above already captures this preference; we just need to specify new linearizations with `not_b` in them. The pseudocode

```
lin NoSourceFile =
  lambda c:
    not_b(asourcefile(lambda x: c(x)));
lin Not_VP VP =
  lambda c:
    not_b(VP(lambda v: c(v)));
```

captures the fact that the negation of "A source file compiles" is not (24) but "No source file compiles".

To generate natural-sounding negations of sentences containing quantification, some simple logical equivalences are necessary. To take an extreme example, suppose that Prograder needs to negate the requirement specification

(27) Every text file mentions no source file.

Strictly speaking, it would be correct to generate

(28) Not every text file mentions no source file.

However, it would be much more comprehensible for Prograder to report instead

(29) A text file mentions a source file.

One heuristic for preferring (29) over (28) is to use as few negations as possible. But to apply this heuristic, Prograder must first realize that (29) is a correct negation of (27). In general, our concrete grammar ought to equate formulas that are equivalent by De Morgan's laws. Unfortunately, GF can no more equate formulas by De Morgan equivalence than by beta-equivalence.

In lieu of equating formulas, we normalize them: we use De Morgan's laws to move negations logically as far "inside" as possible. In other words, we impose an invariant on our semantic representation, that `not_b` applies only to atomic formulas (as in (8–9)). This invariant is easy to enforce in our Python code for gathering evidence, because we can rewrite each evidence statement after generating it. It is trickier to enforce the invariant in our GF concrete grammar for semantic representations, because (to keep parsing tractable) linearizations can only be concatenated, never inspected and rewritten. Therefore, our linearizations must maintain formulas alongside their De Morgan duals.

Specifically, we revise the simulation described in the previous section as follows. The record

(30) `{ spl =` s_l^+`; spm =` s_m^+`; spr =` s_r^+`;`
`snl =` s_l^-`; snm =` s_m^-`; snr =` s_r^-`;`
`switched = False }`

represents a higher-order function of the form

(31) `lambda c:` s_l^+ `c(`s_m^+`)` s_r^+

assuming that it is equivalent to

(32) `lambda c: not_b(`s_l^- `not_b(c(`s_m^-`))` s_r^-`)`

Dually, the record

(33) `{ spl =` s_l^+`; spm =` s_m^+`; spr =` s_r^+`;`
`snl =` s_l^-`; snm =` s_m^-`; snr =` s_r^-`;`
`switched = True }`

represents a higher-order function of the form

(34) `lambda c:` s_l^+ `not_b(c(`s_m^+`))` s_r^+

assuming that it is equivalent to

(35) `lambda c: not_b(`s_l^- `c(`s_m^-`)` s_r^-`)`

For example, given the linearizations

```
lin NoSourceFile = {
  spl = "everysourcefile ( lambda x :";
  spm = "x"; spr = ")";
  snl = "asourcefile ( lambda x :";
  snm = "x"; snr = ")";
  switched = True };
lin EverySourceFile = {
  spl = "everysourcefile ( lambda y :";
  spm = "y"; spr = ")";
  snl = "asourcefile ( lambda y :";
  snm = "y"; snr = ")";
  switched = False };
lin ASourceFile = {
  spl = "asourcefile ( lambda x :";
  spm = "x"; spr = ")";
  snl = "everysourcefile ( lambda x :";
  snm = "x"; snr = ")";
  switched = False };
```

(and an alphabetic variant of `ASourcefile`), Prograder uses the `switched` flags to deduce that one way to negate "Every source file mentions no source file" is "A source file mentions a source file".

Our solution demonstrates that an existing grammatical software package can express continuation grammars. While the record-based implementation is somewhat unwieldy when encoded in the grammar by hand, restricting ourselves to GF's context-free rewrite grammars also ensures efficient parsing.

6 Putting It All Together

Currently, our English grammar supports declarative sentences with a small vocabulary of transitive and intransitive verbs ("exists", "compiles", "has comments", "mentions"), proper noun phrases referring to specific source files, noun phrases representing quantified nouns, and negations.

Given that the grammars correctly specify logical scoping and natural language syntax for parsing and generation, the Python code that implements requirement checking and evidence gathering is relatively straightforward. As described above, the logical form of the requirements specification is executable Python, and their execution emits an evidence statement in the same logical language for each check performed, in a tree structure. Thus the execution of the checking code is an evidence-gathering or proof-search process. The evidence statements are then translated to natural language by means of the two GF grammars.

A Python "glue script" ties the Python and GF components together and manages the dataflow of the end-to-end system. This script provides a simple scanner and symbol table to replace file names with standardized placeholders from the grammar. The variables used in lambda expressions also need to be renamed in order to prevent conflicts.

Here is a sample output of the system as it currently runs:

```
./runPrograder.py 'every source file
    compiles and every source file has
    comments'
*****************************
RESULT: False, because:
some source files don't compile and
some source files don't have
comments:
"nowork2.c" doesn't compile
"nowork.c" doesn't compile
"nowork2.c" doesn't have comments
"nowork.c" doesn't have comments
```

7 Conclusions and Future Work

To our knowledge, Prograder incorporates the first implementation of continuation-based semantics within a grammatical framework that supports efficient parsing and generation. Consequently, our declarative grammar uniformly expresses quantifier meanings and scope preferences.

We want to see how far we can stretch a record-based grammar system such as GF to handle quantifiers and negation using continuations. In the end, the boilerplate ingredients of our solution ought to be automated, so as to combine the expressivity of continuation-based semantics with the usability and efficiency of GF. This will also make it easier to expand the range of natural-language constructs that the system handles. Of course, this development should be driven by feedback from actual instructors using the system, which we also intend to obtain.

Our second area of future work is the summarization of explanations. We plan to use Prograder to investigate the semantics and pragmatics of summarization, and to search for underlying principles based on proofs and types.

References

Alfred V. Aho and Jeffrey D. Ullman. 1969. Syntax directed translations and the pushdown assembler. *Journal of Computer and System Sciences*, 3(1):37–56, February.

Chris Barker and Chung-chieh Shan. 2008. Donkey anaphora is in-scope binding. *Semantics and Pragmatics*, 1(1):1–46.

Chris Barker. 2002. Continuations and the nature of quantification. *Natural Language Semantics*, 10(3):211–242.

Johan Bos. 1995. Predicate logic unplugged. In Paul Dekker and Martin Stokhof, editors, *Proceedings of the 10th Amsterdam Colloquium*, pages 133–142. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Olivier Danvy and Andrzej Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December.

Philippe de Groote. 2002. Towards abstract categorial grammars. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 148–155, San Francisco, CA, July. Morgan Kaufmann.

Pierre-Joseph Gailly. 1988. Expressing quantifier scope in French generation. In *COLING '88: Proceedings of the 12th International Conference on Computational Linguistics*, volume 1, pages 182–184.

Herman Hendriks. 1993. *Studied Flexibility: Categories and Types in Syntax and Semantics*. Ph.D. thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Jerry R. Hobbs and Stuart M. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13(1–2):47–63.

Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys*, 28.

Alexander Koller, Joachim Niehren, and Stefan Thater. 2003. Bridging the gap between underspecification formalisms: Hole semantics as dominance constraints. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, pages 195–202, Somerset, NJ. Association for Computational Linguistics.

Richard Montague. 1974. The proper treatment of quantification in ordinary English. In Richmond H. Thomason, editor, *Formal Philosophy: Selected Papers of Richard Montague*, pages 247–270. Yale University Press, New Haven.

Douglas B. Moran. 1988. Quantifier scoping in the SRI core language engine. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, pages 33–40, Somerset, NJ. Association for Computational Linguistics.

Aarne Ranta. 2004. Grammatical Framework: A type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189.

Stuart M. Shieber. 1993. The problem of logical-form equivalence. *Computational Linguistics*, 19(1):179–190.

Mark J. Steedman. 1996. *Surface Structure and Interpretation*. MIT Press, Cambridge.

Kurt A. VanLehn. 1978. Determining the scope of English quantifiers. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.