SEMINAL: Searching for ML Type-Error Messages

Benjamin Lerner   Dan Grossman   Craig Chambers
University of Washington
{blerner, djg, chambers}@cs.washington.edu

Abstract

We present a new way to generate type-error messages in a polymorphic, implicitly, and strongly typed language (specifically Caml). Our method separates error-message generation from type-checking by taking a fundamentally new approach: we present to programmers small term-level modifications that cause an ill-typed program to become well-typed. This approach aims to improve feedback to programmers with no change to the underlying type-checker nor the compilation of well-typed programs.

We have added a prototype implementation of our approach to the Objective Caml system by intercepting type-checker error messages and using the type-checker on candidate changes to see if they succeed. This novel front-end architecture naturally decomposes into (1) enumerating local changes to the abstract syntax tree that may remove type errors, (2) searching for places to try the changes, (3) using the type-checker to evaluate the changes, and (4) ranking the changes and presenting them to the user.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.4 [*Programming Languages*]: Compilers

General Terms Languages, Design

Keywords Type-Checking, Type-Inference, Error Messages, Objective Caml, Seminal

1. Introduction

The benefits of sophisticated polymorphic type systems (for safe and flexible programming) and type inference (for eliding cumbersome types) hold for programs that type-check. Ill-typed "programs" have no meaning and type-checkers' attempts at producing useful error messages are widely acknowledged as needing improvement. Prior attempts to improve error messages have modified the type-checker to maintain additional information (e.g., constraint dependencies) that is unnecessary for determining if a program type-checks. This approach has several disadvantages:

1. Maintenance: Type-checking algorithms are often elegant before they are modified for error-message generation. The extra information can make type-checkers much more difficult to write and change, impeding type-system improvements.
2. Correctness: Type-checkers are trusted components of most safe-language implementations. Making them more complicated can introduce compiler defects even though error-message content need not be trusted.
3. Speed: Type-checkers must be efficient enough in practice to validate large programs. Assuming a large program contains mostly code that type-checks, the vast majority of time and space spent maintaining extra information is wasted.

A solution for issues (2) and (3) is to maintain two type-checkers — a simple, fast one that makes little effort to isolate the source of ill-typedness and a slower, untrusted one that provides better feedback. However, naïvely following this approach exacerbates problem (1) because we have the difficulties of maintaining a complicated good-message type-checker plus the new task of keeping it consistent with the trusted type-checker. This paper presents a new way to describe type-errors and a new compiler front-end that produces good messages while avoiding the disadvantages just enumerated. Our system is called SEMINAL (Searching for Error Messages IN Advanced Languages).

The new way to describe type-errors is to find a "small change" to the program that results in a similar program that type-checks. For example, if the expression `f x y` does not type-check then perhaps `f (raise DummyExn) y` or `f y x` or `f (x y)` will. This style of error message complements the conventional approach of describing why type inference failed. First, it focuses on terms instead of types, which may better match the programmer's view of the program. Second, it is independent of the type-inference algorithm, allowing it to suggest a change in a program location far from where type inference discovered an inconsistent constraint.

Our compiler architecture has three key pieces: The *changer* takes an ill-typed program and generates small changes that may make the program type-check. The changer itself decomposes into an *enumerator* that suggests local syntax changes and a *searcher* that guides where to attempt changes. The *type-checker* is unchanged but assumes a new role as a decision procedure that determines which generated changes succeed. The *ranker* prioritizes successful changes and displays messages accordingly. Key to the approach's maintainability is that the changer and ranker are untrusted and do not assume anything about the type system (i.e., one could change the type system without changing them).

The compile-time computational cost of calling the type-checker many times with slightly different programs is justified on two grounds. First, we do so only for ill-typed files; the approach adds zero cost to type-checking well-typed files. Second, we assume the consumer of our error messages is a human. People are slow and expensive: if a couple seconds at compile-time can produce a useful suggested change that the human would have found in a few minutes, the computation is well worth the cost.

We have built a prototype implementation of our approach for Objective Caml and consider it a work-in-progress. As hoped, we

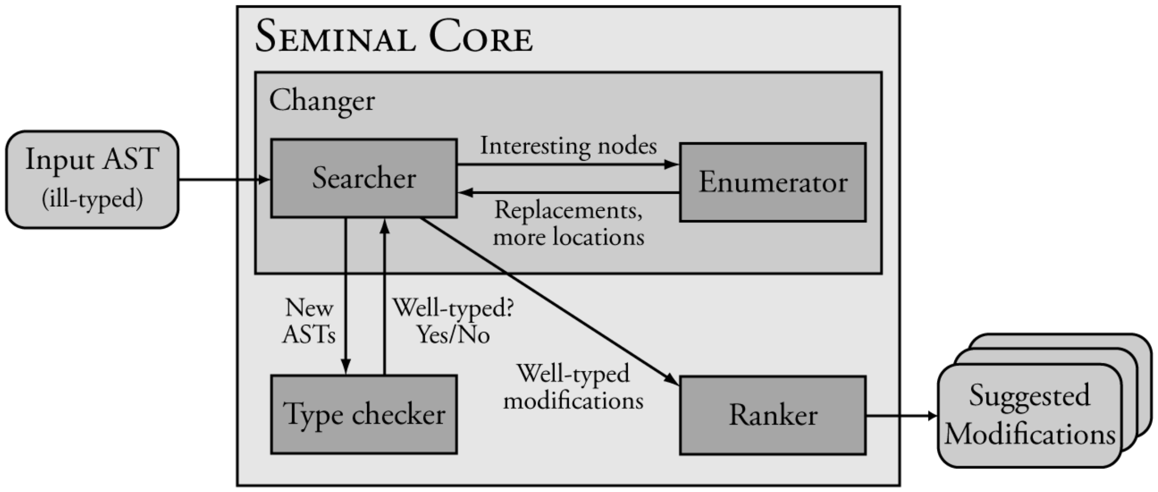

Figure 1. The SEMINAL architecture

have made no changes to the type-checker. Instead, we implement a "wrapper" in the compiler that intercepts the type-checker's error messages and tries to produce better ones. The implementation and its interaction with the type-checker are made easier by ML being the compiler-implementation language and the language-being-compiled, but we hope the approach applies broadly.

The rest of this paper is organized as follows. Section 2 gives an overview of our system and some examples of the sort of changes we consider. Sections 3, 4, and 5 describe the enumerator, searcher, and ranker respectively. Section 6 extends our approach with *dependent modifications*, which lead to a more elegant and efficient interaction between the enumerator and the searcher. Section 7 describes the current status of our prototype and our plans for future work. Section 8 discusses related work, and Section 9 concludes.

2. Overview

SEMINAL's error-message generation operates on untyped abstract syntax trees (ASTs). That is, the code we add to the compiler front-end sits between parsing (which converts a token stream to an untyped AST) and type-checking (which converts an untyped AST to a typed AST). An AST provides much more structure than a token stream, which makes it easy to search for relevant places to make changes to the program (Section 2.2). An untyped AST lets us enumerate slightly different ASTs without needing to produce types (Section 2.3).

2.1 Compiler Architecture

Figure 1 shows how the pieces of our implementation fit together. Initially, the AST resulting from parsing is passed to the type-checker unchanged. If type-checking succeeds, compilation proceeds unchanged. Otherwise, the type-checker raises an exception. SEMINAL intercepts this exception and transfers control to the changer.

Conceptually, the changer searches for places in the AST to make changes and enumerates a collection of possible changes at all such places. It then calls the type-checker with the AST corresponding to each change, discards the ASTs that do not type-check, and passes the remaining ones to the ranker for presentation to the programmer.

In practice, searching, enumeration, and type-checking interact at a finer level. First, the search itself uses the type-checker to avoid enumerating changes where it is fruitless (Section 2.2). Second, we type-check each change as soon as it is enumerated. This approach is more efficient and allows *dependent modifications* (Section 6) in which the success of one change guides whether we attempt other changes. One could also imagine interleaving ranking with searching and enumeration (i.e., not try changes if better changes have already succeeded), but we do not currently do so.

2.2 Searching the AST

Given an AST that does not type-check, we wish to identify "interesting" nodes in the tree, i.e., nodes where we should enumerate local changes to produce new ASTs. Treating every node as interesting is too expensive since there are many nodes and we are not shy about enumerating many changes for each interesting node.

A simple top-down procedure can quickly eliminate most nodes from consideration. Starting at the root, replace a node with a "wildcard" (e.g., an expression like `raise DummyExn` that can have any type) and pass the resulting program to the type-checker.[1] If the result type-checks, deem the node interesting and recur on its children to find more interesting nodes. If not, deem the node and all its descendents uninteresting.

This procedure has several nice properties. First, it always finds interesting nodes (e.g., the root is always interesting). Second, it does not recur into subtrees where no change can fix the type error. Third, the type-checker gives useful information about the wildcard replacement: the type inferred for the wildcard is something we can report to the user without the searcher understanding the type system. (This information is already computed by the type-checker but unnecessary for our search procedure; it is essentially a "bonus" from SEMINAL's perspective.)

Fourth, the procedure is not limited to one path in the AST (so given `let x =` e_1 `in` e_2, it can find interesting nodes in e_1 and e_2), and it is not constrained by type inference. An alternate search procedure might start at the node n where the type-checker reported an error and consider ancestors, descendents, and neighbors of n. We believe doing so restricts SEMINAL counterproductively, since it is well-known that type inference can identify locations that are arbitrarily far from the error source.

This search procedure has limitations. Most importantly, it does not do well with programs that have multiple independent type errors. If two typing problems in the source program require two separate changes, then only a shared ancestor of the changes is

[1] In languages without expressions that have any type, interacting with the type-checker would require additional work.

```
print_string "a" ^ "b"    print_string "a"                       print_string ("a" ^ "b")
                          This expression has type               print_string "a" [[..infix..]] "b"
                          unit but is here used                    with [[..infix..]] : unit->string->'a
                          with type string                       [[...]] "a" ^ "b"
                                                                   with [[...]] : string->string

sumList [5, 6, 7]         5, 6, 7                                sumList [5; 6; 7]
                          This expression has type               sumList [[...]]
                          int * int * int but is here used         with [[...]] : int list
                          with type int                          [[...]] [5, 6, 7]
                                                                   with [[...]] : (int*int*int) list -> 'a

match x with              A3                                     match x with
  | A1 -> 1               This pattern matches values of type      | A1 -> 1
  | A2 -> match y with    a but is here used to                    | A2 -> (match y with
    | B1 -> 11            match values of type b                     | B1 -> 11
    | B2 -> 21                                                       | B2 -> 21)
  | A3 -> 5                                                        | A3 -> 5
```

Figure 2. Three examples of type errors, OCaml error messages, and some suggested changes

deemed interesting. Handling multiple errors is important future work, but we mitigate the problem by considering each top-level binding separate from what follows it in a module (just as the ML type-checker does).

2.3 Enumerating Other ASTs

Sometimes identifying an interesting node with the procedure above is all we need to do. For example, for the trivial program

```
let a = "hello" in
(a+3)*(a+2)
```

the search procedure determines that replacing the expression `"hello"` with an `int` suffices for type-checking. (It also determines that replacing `(a+3)*(a+2)` with any well-typed expression suffices, but the ranker prefers the former, smaller change to this larger one.)

Other times, this sort of "wildcard replacement" (replacing an expression with one of any type) is a less informative suggested change. For the program

```
let f x y = print_string (x ^ (string_of_int y))
in f 3 "hello";
   f 4 "world"
```

a good change is to replace `f x y` with `f y x`, yet no wildcard replacement comes close to approximating it.

Therefore, for each interesting node, we enumerate all local changes to the AST and see if the resulting new trees type-check. The changes we consider depend on the particular AST node type. For the example above, function parameter lists lead to possible changes including parameter rearranging (as above), parameter removal, and parameter addition (adding a wildcard argument).

While developing our collection of local changes (discussed more completely in the next section), we have been guided by a realization that a given change can be more or less justified in terms of a tree edit. For example, swapping function parameters is just swapping the order of children at an AST node, a simple operation worth trying at many nodes. Other "tree-based" changes include rotations (which often resolve precedence errors) and additions (which can resolve partial-application errors).

At the other extreme, some local changes account for peculiarities of the source language, i.e., they are not derived from first principles of trees. As an example, `[1,2,3]` is a list containing one triple, so we try `[1;2;3]`, a list of three integers. In the AST, this change involves deleting a node (for the triple) and making its children be children of its parent. So it is "tree-based," but we do not believe this transformation (move children up one level) is worth applying in general.

By separating the enumerator in our architecture, adding new possible modifications for a particular sort of AST node requires just a couple lines of code. Also, completeness is not a major concern because the lack of highly-specific modifications hurts only possible error messages for that node, and we can always fall back to reporting a wildcard replacement or the underlying type-checker's error message. For example, we have little experience with Caml's object-oriented features, so we do not yet enumerate any changes for them.

2.4 Examples

Figure 2 shows three snippets of ill-typed programs, the node and error message reported by the Caml type-checker, and the suggested changes reported by SEMINAL. The syntax `[[...]]` is just how we print `raise DummyExn` since the latter is an implementation trick. These examples demonstrate situations where SEMINAL benefits from a global search (rather than focusing on where type inference failed) and specific term-level changes (rather than focusing on the type system). The first suggestion in each case is a specific change. Wildcard-replacement messages, which show the type inferred for the wildcard, are as informative (though perhaps less straightforward) as the OCaml error message.

3. Enumerating Modifications

Every AST modification replaces a subtree with a new one, in the hope that the resulting AST typechecks. For any subtree, one simple approach is to replace the subtree with a "wildcard", a tree that imposes no type constraints. For example, an expression subtree should be replaced by one of type α. This replacement's generality makes it widely applicable, but more refined modifications consider the structure of the node being changed. As such, enumerating modifications is easily organized by type of AST node; `let`-nodes suffer from different errors and can be resolved by different modifications than `match`-nodes, for instance.

Figure 3 summarizes several modifications that SEMINAL attempts. Section 3.1 considers in more detail how we identify useful changes at individual syntax nodes. Section 3.2 explains how our framework addresses the problem of nested `match` expressions. Section 3.3 discusses how we define modifications in our implementation.

Generic replacements		
`e1`	`raise DummyExn`	Replace any expression with a wildcard of type 'a
`p1`	`_`	Replace any pattern with a wildcard of any type
`t1`	`'a`	Replace any explicit type with a fresh type variable
Identifiers		
`id1`	`id2`	Try another variable in scope
`r.fld <- e`	`r.otherfld <- e`	Try another field name in scope
Function applications		
`fExp a1 a2 a3 ... an`	`fExp a1 a2 (raise DummyExn) a3 ... an`	Insert an arbitrary new argument
	`fExp a1 a3 ... an`	Delete an extraneous argument
	`fExp a3 a2 a1 ... an`	Swap any two arguments
	`fExp a2 a3 a1 ... an`	Move a single argument to a new location
	`fExp (a1 a2 a3 ... an)`	Reassociate function arguments
	`fExp (a1, a2, a3, ... an)`	Uncurry function arguments into a tuple
`fExp (a1 a2 a3 ... an)`	`f a1 a2 a3 ... an`	Unassociate function arguments
`fExp (a1, a2, a3, ... an)`	`f a1 a2 a3 ... an`	Curry tupled arguments
`r.fld := e`	`r.fld <- e`	Mutable binding instead of field or array assignment
Let bindings		
`let p1 = e1 in e2`	`let rec p1 = e1 in e2`	Toggle the `rec` flag to mark a self-referential `e1`
`let p1 = e1 in let p2 = e2 in e3`	`let rec p1 = e1 and p2 = e2 in e3`	Hoist nested lets into one `let rec`
Match expressions		
`match e with p1 -> e1 | p2 -> ...`	`match e with p2 -> e2 | ...`	Remove a single match case
	`match e with ... | _ -> raise DummyExn`	Add a catch-all case at the end
`match e1 with` `| p1 -> match e2 with` `| p2 -> e3` `| p3 -> e4` `| p4 -> e5`	`match e1 with` `| p1 -> (match e2 with` `| p2 -> e3` `| p3 -> e4)` `| p4 -> e5`	Reassociate inner cases of match expressions outward.
Miscellaneous		
`[x, y, z]`	`[x; y; z]`	Convert a list of a single tuple into a list of the tuple's members
`let p : t1 = e1`	`let p = e1`	Remove unnecessary type constraints

Figure 3. Selected modifications that SEMINAL tries for various expression forms

```
type astThing = Exp of Parsetree.expression
              | Pat of Parsetree.pattern
              | Typ of Parsetree.typ
              | ...
type astRepl  = { revision : (unit -> astThing);
                  tag      : int;
                  loc      : Location.t;
                  validate : (unit -> bool);
                  ... }
```

Figure 4. Record type defining SEMINAL modifications; some details omitted

3.1 Simple Changes

Consider the Caml expression `fExp a1 a2 a3 ... an`. Several kinds of errors are specific to this expression: arguments to the function might be missing or extraneous; arguments might be in the wrong order; because of operator precedence the expression might be mis-parsed; arguments might be tupled instead of curried; etc. Other errors are not endemic to this particular node: any of the subexpressions `fExp`, `a1`, ..., `an` might itself be erroneous. We therefore consider them separately, when later examining their respective nodes. To determine what modifications to try for a given AST node, it is useful to organize modifications into two broad categories: principled, tree-based modifications, and language-specific, expert-knowledge modifications.

The appropriate repairs for several of these problems, such as missing or extraneous arguments, or mis-parsed precedence among arguments, are natural *tree manipulations*. For instance, mis-parsed infix operator precedence is resolved by a tree rotation: compare the ASTs resulting from the type-incorrect parsing `(print_string "hello") ^ "world"` versus the intended `print_string ("hello" ^ "world")`. Tree manipulations generally include node deletions, node insertions, and node rearrangements [1, 4, 17]; we also found subtree insertions and deletions and node rotations useful. Tree manipulations are easy to enumerate, since there are a fixed number of operations, and a finite-sized node on which to apply them. In practice, when implementing the modifications applicable to a new AST node, we start by considering the ways a tree manipulation could fit the new node, and then implementing the relevant ones.

The remaining problems, such as incorrectly tupled or curried arguments, are less usefully described by their associated tree manipulations. As the SEMINAL implementors, we conceive of currying tupled arguments, and then code it up by removing an AST node (for the tuple) and making its children be children of its parent (the application node). Such modifications are motivated by specific knowledge of Caml and its syntax. Expert knowledge of common mistake patterns is necessarily "incomplete," since everyone makes different kinds of errors. However, SEMINAL does not require exhaustive knowledge to be useful, and it easily supports incremental improvements via the definition of new modifications.

Another "expert-knowledge" example arises with `r.fld := v` where `r.fld` is a mutable field of the same type as `v`. If the intent was to mutate the record's field, the correct code is `r.fld <- val`. OCaml implements the first (assignment to a reference) as a library function, while the second (field mutation) is special syntax. As such, the tree modification turns a function application into an assignment when the function argument is a particular library function (`:=`). While easy to implement, generalizing this tree modification is not useful.

3.2 Nested Match Changes

A common mistake with nested match expressions is forgetting to close the inner match so that subsequent cases are grouped with the outer match. If the patterns in the cases have different types, a type conflict occurs. This error is an artifact of Caml's concrete syntax: there is no "end-match" token, so the ambiguity is akin to the dangling-else problem, and is resolved in the same way. Similar ambiguities arise with `try...with...` and `fun...` expressions; for simplicity we illustrate the problem with `match` expressions.

Resolving a nested-match error requires modifying the structure of the expression's subtree, including nodes for an arbitrary number of nested matches. This operation is more sophisticated than the other modifications in Figure 3. Nevertheless, the modification is still "local," since it changes only the nodes for the nested match expressions. When SEMINAL encounters an "interesting" match expression, it recursively gathers into a flat list all the cases and match tests from all the nested matches. The goal is to recreate the possible nesting structures which could have produced that list, hoping that one of them—the one with the correct association of cases to matches—type checks.

Once the match cases have been gathered, SEMINAL could construct a massive list of modifications, where each one suggests an alternate way of grouping the internal match cases. In the example shown, SEMINAL considers three regroupings: that the `B1` case alone be grouped with the `match y` expression, that `B1` and `B2` be, or that `B1`, `B2`, and `A3` be. In this example, where we assume that the `An` and `Bn` patterns match distinct types, only the second modification type-checks, but the enumerator has no understanding of the type system. In general, patterns can match values of multiple types, so multiple solutions might be found.

The algorithm just described is simple, but suffers from severe performance problems: there are simply too many regroupings of nested expressions. In the simplest case, where all the nesting occurs in the very first cases of each nested subexpression, the problem is one of choosing m places to put close-parentheses from a list of n match cases, where m is the number of nested match subexpressions. (Recall "n choose m" is exponential in m.) In other cases, when cases and match subexpressions interleave, the total number of ways can be exponential in the number of match cases, which in practice is much larger.

We do not actually compute every possible regrouping eagerly. Instead we try to parenthesize just a prefix of the cases and use these intermediate modifications to guide ones that parenthesize more cases. Doing so requires modifications that can enumerate more modifications, which is the key generalization that we defer to Section 6.

3.3 Implementation of Modifications

The task of enumerating the set of modifications applicable to a particular AST node is conceptually defined by a decomposition routine, `enumerate : astThing -> astRepl list`.[2] Specialized for each syntactic category (expressions, patterns, types, etc.), this routine simply defines a catalogue of possible modifications for each AST node type.

The type defining a modification is partially shown in Figure 4. The key field is the `revision` thunk, which evaluates to a new AST subtree with which to replace an older one. For instance, the `astRepl` record that defines the generic "replace expression with α" modification contains a thunk that returns the AST for `raise DummyExn`. Similarly, the record that defines the "change mutable binding to field assignment" modification uses the subexpressions of `r.fld := e` in its `revision` thunk to produce the AST for `r.fld <- e`. The enumerator, however, knows nothing about the

[2] Later sections change the return type.

surrounding AST in which this subtree exists. It relies on its caller to retain that context, within which to place the new subtree so as to rebuild the entire AST. This context is maintained by the searcher (see Section 4).

We construct the replacement AST lazily to improve space and time performance. Many modifications come in groups, such as inserting an extra argument into a function application: the same operation can be performed before or after any of the existing arguments. This commonality lets us share values (the original argument list and the inserted argument) until actually computing the distinct replacements (constructing the new argument list with the inserted argument at a unique position). We can then reclaim the space sooner: it is unlikely that all the proposed modifications will succeed, so the failed ones can be discarded quickly. The time-performance reasons arise in Section 6; in essence we extend the laziness of computing replacements to the laziness of evaluating them as well.

The remaining fields of the `astRepl` assist ranking (for `tag` and `loc`, see Section 5) and dependent modifications (for `validate`, see Section 6).

4. Searching for Modifications

The goal of the searcher is to identify AST nodes where enumerated changes may succeed. In this section, we summarize our algorithm for finding such nodes, qualitatively evaluate its effectiveness, and describe its implementation in terms of *contexts* as often seen in language semantics.

4.1 Search Locations

Since the goal is to eliminate from consideration as much of the AST as possible, SEMINAL takes a top-down approach to examining the AST, looking for the "interesting" nodes as sketched in Section 2.

We first find the earliest top-level binding that does not type-check. In ML, subsequent bindings cannot be relevant so we remove them (i.e., we truncate the file).[3] In our current prototype, we also assume the preceding top-level bindings have the correct types. By caching these types (essentially creating a signature for them), we can avoid type-checking them more than once. Hence our repeated calls to the type-checker involve exactly one top-level function or expression.

Within this top-level `let`-binding, we start by replacing the entire bound subexpression with a wildcard. This change eliminates the error, so we recursively descend deeper into the AST, to try removing subexpressions of the parent and seeing if those changes also resolve the error.

Conceptually, the search routine maintains a worklist of interesting nodes at which to search. When removing a node from the worklist, it uses the enumerator to list local modifications as described in the preceding section. In practice, the enumerator also returns the node's children because this information is easy for the enumerator—which is already specialized for each AST node—to determine. If replacing a node with a wildcard fixes the type error, then the searcher tries the other modifications and places the node's children on the worklist. Otherwise, the search continues with the rest of the worklist.[4] All successful modifications, including wildcard replacements, are retained for the ranker.

[3] This simplified approach does not handle details of signature ascription. A full solution must first remove the signature, and then proceed to truncate the bindings as above.

[4] There are additional interactions possible between modifications, which can improve search efficiency; see Section 6.

4.2 Effectiveness of Search

This top-down search ensures that SEMINAL finds the smallest expressions that can be changed to cause the program to type-check. This technique also prunes irrelevant side branches efficiently, since unsuccessful wildcard replacements eliminate uninteresting nodes quickly, high in the AST.

The most striking feature of this algorithm, however, is its complete independence from the type checker's inner workings. Normally, the type checker produces some location at which it reports the error, and this location can be unintuitively far from the actual cause of the problem. SEMINAL is not guided by these locations, however, and might instead find the actual problematic location. Moreover, the type checker could be changed drastically (algorithm $\mathcal{M}$ replaced with algorithm $\mathcal{W}$, or with a non-Hindley/Milner algorithm for a different language) and SEMINAL would still find interesting nodes to replace and modifications to suggest.

While this algorithm works well in practice, there are important extensions that would improve its explanatory capabilities. SEMINAL's algorithm constantly looks for *local* changes to each node. By delegating the responsibility for decomposing each node onto the enumerators, it relies on the enumerator to list the relevant locations of its children, for use with future searching. This locality is an artifact of the datatype used to represent the AST.

If the datatype cannot represent certain relationships in the code as a "local" property of the nodes, SEMINAL cannot take advantage of it to find more effective locations to search. Foremost among these useful relationships is the ability to jump from an expression using an identifier to the defining expression for that identifier, in the hope that if replacing one use might resolve one error, changing the definition might resolve errors at multiple uses later in the file.

Generalizing this problem, we see that some expressions cannot be repaired except with multiple, coordinated modifications. Consider `let x = 1.0 in (x+1) +. (x+1)`. Either the definition of `x` and the floating-point addition must be simultaneously replaced, or the integer additions with 1 must be simultaneously replaced. Currently, SEMINAL can suggest only `let x = 1.0 in raise DummyExn` as an appropriate change. When multiple coordinated changes are necessary, SEMINAL will currently produce wider error locations than the underlying type-checker.

4.3 Implementation of Search

The key role of recursively descending into the AST is to narrow the scope of the changes to the most precise region of code that can be modified to resolve the type error. This requires that each recursive step maintain context describing the rest of the AST surrounding the active subtree. To do so, we decompose the program into two pieces that can be reconstituted into the entire AST: A subtree of the AST and an AST with a "hole" in it. The latter is represented by a function that takes an AST, fills the hole with it, and returns the resulting complete AST. That is, we can represent an "interesting place" via a pair of this type:

```
    astThing * (astThing -> astThing)
```

This approach is essentially a use of *contexts* as often seen in programming-language semantics. Our notion of an active context is one in which the active node is interesting and is being examined for possible modification. For example, suppose the second element of a tuple `(e1, e2, e3)` is interesting. SEMINAL essentially describes this context as the pair `(e2, (fun x -> (e1, x, e3)))`, where by `(e1,x,e3)` we actually mean the AST for a triple built from the ASTs bound to `e1`, `x`, and `e3`. This encoding of contexts permits a clean and uniform search, regardless of the type of AST node being examined.

Descending from a parent to a child in the AST requires composing a "global context" (the context that takes the parent's replacement and produces a new program) with a new "local context" (the context that describes how to rebuild the parent node after changing the child node). The searcher's conceptual worklist has items of type `astThing * (astThing -> astThing)` where the latter is the global context for the item and the former is passed to the enumerator. The enumerator returns a list of new ASTs to "plug into the hole" *and* a list of child nodes to visit recursively (if doing so is useful).[5] For elements of the latter, the enumerator returns an `astThing * (astThing -> astThing)` where the `astThing` is a child node and the function describes only how to recreate the parent node given a new child node. This function is the "local context" for the new worklist item, but the searcher can create the child's "global context" via function composition with the parent's "global context".

In practice, we maintain for each worklist item the "global context" (which allows us to create an AST for passing to the type-checker) and a "sliding window" of the 5 most recent "local contexts". These local contexts are useful for printing error messages that include some surrounding context. When recurring in the AST, the sliding window simply discards the fifth most-distant local context and replaces it with the local context returned from the enumerator. When a modification succeeds, we return a "fairly local context" built from the information in the sliding window.

As a final detail, we do not actually maintain an explicit worklist, relying instead on the call-stack implicit in our recursive search procedure.

By using contexts and having the enumerator generate local contexts, our entire search procedure (including the extensions in Section 6) is small enough that it appears (with very modest simplifications) in the Appendix.

5. Ranking Successful Modifications

Given a collection of successful changes, we must decide which ones to present to the programmer and in what order. Choosing the "best change" is inherently ill-defined; the source program is semantically meaningless and we do not know the programmer's intent. Nonetheless, presenting all successful changes in arbitrary order is unacceptable: There are typically many and SEMINAL's goal is to produce a concise, useful error message. Therefore, we combine heuristics and explicit guidance from the enumerator to rank the changes.

Three techniques filter out changes where a "clearly better" change is also successful. The first exploits a general property of trees; the others encode knowledge of our *ad hoc* changes.

- "Smaller" changes are better than "larger" ones: If one modification changes code that is strictly contained in code changed by another modification (a subtree), we discard the latter in favor of the former. Moreover, a modification that deletes an entire subtree is a "larger" change to the AST than one that modifies just the root node of that subtree, even though they span the same region of code. The `loc` field in Figure 4 enables this filter.
- Enumerated changes can specify a preference order: We can specify that one local change to an AST node is always preferable to another. For example, reparenthesizing match expressions is preferable to deleting cases from a match expression. Specifying a preference is simple using the `tag` field in Figure 4.
- Enumerated changes can specify whether they should be displayed at all using the `validate` field in Figure 4. As we will show, the intermediate steps in computing a reparenthesized match expression are each legitimate modifications, but should not be presented to the user.

These filters rarely conflict, but we apply the latter ones first since they account for specific sorts of changes.

We then rank the remaining changes. We give "shorter changes" higher rank given some metric for defining distance. At present, we have implemented three such metrics:

- Textual length of code. This metric is the simplest, but least accurate, as source-code comments affect a change's length.
- String-edit distance [20] of pretty-printed code. This metric is similar but slightly more robust, as it normalizes the text into some standard form. However, it too can be biased: longer identifiers increase length.
- Tree-edit distance [1] of the two ASTs. This metric is the most expensive to compute (and for large trees we skip it), but produces a metric robust to the problems of the first two. It suffers from the reverse problem, of not being sensitive enough to textual changes in the code.

In practice, we suspect that a combination of these metrics will yield the best results. We are still experimenting with these metrics, so we have implemented SEMINAL such that it is trivial to replace or combine distance metrics.

After ranking, we currently print the changes in ranked order. In future work, we intend to address relevant user-interface issues. For example, is it better to print only n changes, perhaps providing a way to see more ("click here for more suggestions")? Also, can the programmer provide feedback about which change was best (perhaps implicitly via subsequent editing)? Can we adapt the ranker to programmers' preferences?

6. Dependent Modifications

As described so far, SEMINAL tries every modification suggested by the enumerator for every interesting node. In fact, the "things to try" that the enumerator returns are structured such that the searcher tries some later modifications only if some earlier modifications succeed. This generalization has two benefits: it can encode the procedure for finding interesting nodes (Section 6.1), and it permits an efficient algorithm for finding nested-match error (Section 6.2).

To support dependent modifications, we allow one list of modifications ℓ_2 to depend on another ℓ_1 as follows: The modifications in ℓ_2 are attempted only if there exists a modification in ℓ_1 that succeeds. To extend this notion arbitrarily deeply, we use the `astMod` type defined in Figure 5; the enumerator then returns a value of type `astMod list`. In this list, values of the form `Recur v` are child nodes and their local contexts, as explained in Section 4.3. Values of the form `Replace v` contain an `astRepl list` containing modifications that will be tried (an ℓ_1) and a computation to produce another `astMod list` (the ℓ_2) should any of them succeed.

With this generalization, we may wish to try modifications for the sole purpose of determining whether it is worthwhile to try other modifications. In this case, a modification's success should not be passed to the ranker. The `validate` field in `astRepl` encodes this distinction: If a modification succeeds, the thunk in `validate` is called and only if it returns true is the modification given to the ranker.

6.1 Finding Interesting Nodes, Revisited

Recall our top-down algorithm for finding interesting nodes, i.e., nodes where we try enumerated changes and recur on the children: We see if replacing the node with `raise DummyExn` succeeds and try other modifications only in this case. Now with our `astMod`

[5] Actually, the two lists are merged and a datatype distinguishes the two sorts of elements.

```
type astMod =
   Recur of astThing * (astThing -> astThing)
 | Replace of astRepl list * (unit -> astMod list)
```

Figure 5. Datatype defining dependent modifications

definition, we can describe this by having the enumerator return only one modification ("replace with `raise DummyExn`") but with all the other modifications depending on it.

For example, consider three modifications we might try for a pair `(e1,e2)`: recur on `e1`, recur on `e2`, and swap the children (replace `(e1,e2)` with `(e2,e1)`). The case of the enumerator for tuples would make an `astMod list` (call it ℓ_2) of length 3 with these modifications. The enumerator then wraps this list by creating:

`Replace([makeDummyRaise()], fun () ->` ℓ_2`)`

where `makeDummyRaise()` creates the wildcard replacement. This wrapping step can be centralized because we do it for all expressions.

This technique *encodes* the fact that the searcher should recur on a node's children or try other modifications, but only if there is some way that changing the node can cause the program to type-check. Using this encoding simplifies the rest of the searching algorithm.

6.2 Nested Match Changes, Revisited

As discussed in Section 3, the number of ways a nested match expression could be re-parenthesized can be exponential in the total number of match cases. However, with dependent modifications we can encode an algorithm that builds up partial solutions in stages. The algorithm runs efficiently in practice and always finds a full parenthesization that type-checks if one exists.[6]

The key idea is to try to find placements iteratively for the n^{th} case among the available match expressions, but only after finding a placement for the first $n-1$ cases that type-checks. When placing the n^{th} case, there are at most m possibilities where m is the number of match expressions that have not had their "close parentheses" used when placing the first $n-1$ cases. (There may be multiple ways to organize the first $n-1$ cases so they type-check, each of which would lead to trying more modifications to place the n^{th} case.) While this algorithm sketch is admittedly subtle, the interesting point in terms of SEMINAL is that we can implement it efficiently with dependent modifications.

In terms of dependent modifications, the enumerator *lazily* builds an entire tree of dependent modifications where each length-n path from the root corresponds to a particular placement of the first n cases. The laziness is crucial for not investigating any subtrees where we can determine just from the first n cases that the grouping cannot type-check. In practice only very few groupings of the first few cases will type-check, so our algorithm does not visit most of this conceptual tree.

To actually generate an AST modification for the first n cases, we simply truncate the remaining cases (but add cases of the form `| _ -> raise DummyExn` as necessary to ensure matches are exhaustive).

If we find a modification that uses all the original match cases and type-checks, we report this success to the ranker. However, none of the intermediate points where we match fewer cases should be reported. The `validate` field suffices to make this distinction.

[6] Pathological cases exist where many intermediate stages could be parenthesized in many different ways, but such situations require many patterns that have the right type for multiple match expressions.

Test (error type)	# Nodes explored	# Calls to TC	Time (sec)
Function application	27	111	0.230
Argument currying	34	131	0.138
Field access	5	8	0.020
Missing function arguments	11	24	0.072
Reaching definitions	19	52	0.134
Function rotations	15	58	0.117
Tuple-to-list errors	21	79	0.144
Single nested-match error	37	154	0.456
Multiple nested-match errors	29	516	1.744
Datatype errors	41	88	0.259
Seminal itself	81	268	2.664

Figure 6. Runtime performance for several test cases

7. Current Status and Ongoing Work

As currently implemented, our prototype is both fast enough and accurate enough to be useful for many common errors. This section presents quantitative experiments that verify feasibility and identify bottlenecks, discusses qualitative evaluation that is still in progress, and identifies important areas of future work.

7.1 Current Quantitative Results

We built SEMINAL by taking OCaml version 3.08.4, adding one 3700-line (heavily commented) module, and making a few tiny changes in other files (such as adding a command-line option and intercepting exceptions thrown by the type-checker). Our performance results are for running this modified compiler (itself compiled to native code) on a 2.8GHz Linux workstation.

Figure 6 describes the run-time for SEMINAL on several files with type errors. The first column is the number of AST nodes that are changed at any point. (These are interesting nodes and nodes deemed uninteresting after replacing them with `raise DummyExn` is unhelpful.) The second column is the number of calls to the type-checker; it is greater than the first column because we try multiple modifications at some nodes. The third column is total wall-clock time, including parsing. (We do no code generation since the programs do not type-check.)

The tests are grouped by relative complexity: the first set contains short unit tests that illustrate a particular type of modification; the middle set contains two thirty-line programs with different match-nesting complexities; and the final set shows a moderate (130 line) and a long (3700 line) test (the latter being SEMINAL itself seeded with a single typing error near the end of the file). In all cases, SEMINAL produces short and accurate error messages, but of course we created the files ourselves (see below for more objective in-progress evaluation).

Overall, the results are encouraging. None of the tests take more than a couple seconds, which we feel is reasonable for feedback directed to humans. Moreover, our searcher explores very few AST nodes, even for large files.

We also instrumented SEMINAL to determine where the time is spent. It turns out the vast majority of the time is in the type-checker — each call to the type-checker takes a few milliseconds (but much longer for a large file without the optimization described below). That is, SEMINAL's algorithms are not the bottleneck, but calling the type-checker too many times is. To improve performance, we should spend more time in SEMINAL deciding what changes to try rather than simply trying more changes. Put another way, we can try a few thousand modifications before compile-times exceed a few seconds.

SEMINAL's performance on large files requires avoiding unnecessary re-typechecking. In particular, re-typechecking all the toplevel bindings preceding the one with the type error is wasted work—doing it would increase the time for our largest experiment by about a factor of 8. While it is probably not difficult to "checkpoint" the type-checker state at the point after these toplevel bindings have been processed, we did not actually modify the type-checker to do so. Instead, we extract the top-level bindings that type-check, put their types in a signature, and type-check the "problem binding" assuming this signature. A more fine-grained ability to checkpoint should improve performance considerably, but would require some changes to the type-checker.

While SEMINAL is currently efficient for certain large files with certain errors, it is possible to creates files that lead to several thousand attempted modifications. (In particular, errors nested within large nested matches lead to too much wasted work in our current implementation.) As described above, we believe additional caching of type-checker state will solve this bottleneck. We also believe that SEMINAL runs that take longer than a few seconds may still prove worthwhile for programmers, especially if run in the background.

7.2 Future Evaluation

While the quantitative results above show our implementation is feasible, it does not describe the utility of our approach for programmers. To address this, we are pursuing two sources of feedback: a post-facto analysis, using SEMINAL, of real users' errors; and a user study to solicit their real-time opinions while using SEMINAL.

We are currently collecting data from thirteen volunteers who are students in our Master's level course on programming languages.[7] These volunteers have a background particularly interesting for pedagogy and functional programming: They are all full-time software developers, but they do not have prior experience with an ML-like language. These students are using a modified Caml compiler that saves a copy of every file that is passed to the compiler whether the file type-checks or not. Timestamps preserve the files' creation order.

So far we have collected almost two-thousand files, though we expect the data is largely repetitive. (To encourage participation, we promised not to look at the files until the course ends in mid-June.) We will run SEMINAL on each file that does not type-check to estimate how useful it would have been in identifying the students errors.

This off-line analysis of collected files cannot reproduce the experiences of real users, so we are designing a study to examine the reactions of users to SEMINAL's suggestions. In particular, we intend to run a controlled study that evaluates different ranking strategies and SEMINAL's messages compared to the underlying type-checker's messages.

7.3 Future Algorithmic Improvements

In addition to our ongoing work on improving our implementation in natural ways (making it more efficient, adding new modifications, etc.), we have five more substantial additions we intend to pursue soon.

First, we would like to be able to provide better results when a top-level binding suffers from multiple independent type errors. To do so, we need to extend our search to support multiple simultaneous "wildcard replacements". This problem is made more difficult by not knowing how many such replacements to allow, and we believe there are connections to clustering problems where the number of clusters is unknown. It is also possible the programmer could help guide this process.

[7] http://www.cs.washington.edu/education/courses/csep505/06sp/

Second, we would like to support searching via nodes that are conceptually close neighbors even if they are not close neighbors in a particular program representation such as an AST. Use-def chains are probably the most important example.

Third, we would like to use our dependent modifications more aggressively to improve performance. Based on our experience with nested matches, we believe we can reduce the number of modifications we try for function applications by sequencing the changes more carefully. As a simple example, if changing `f x y` to `(raise DummyExn) x y` does not work, then `f y x` will not work.

Fourth, we would like to integrate ranking into the search procedure to reduce the number of calls to the type-checker. That is, we believe it is often more efficient to determine a modification is dominated by an already-successful one that it is to see if the modification will succeed.

Finally, we would like to be able to combine enumerated changes automatically to create larger changes. For example, if the enumerated changes try replacing `(e1,e2,e3)` with `(e2,e1,e3)` and `(raise DummyExn,e2,e3)`, SEMINAL will not try `(e2,raise DummyExn,e3)` unless that is also an explicitly enumerated change.

8. Related Work

Prior work has taken several approaches to improving type-error messages in the presence of type inference. Systems can differ in how they analyze the ill-typed program and in how they present results to the programmer. Unlike SEMINAL, all work we are aware of modifies the underlying type-checker or changes the definition of what should type-check.

Following Heeren's excellent summary [11], we can loosely group prior work into: systems changing the order of unification [14, 15], explanation systems [2, 13], reparation systems [9, 12, 16], program slicing systems [6, 10, 19], and interactive systems [5, 18]; Heeren's own approach fits within the first two categories. Our work is closest to the reparation systems because we suggest modifications to the source program that eliminate the type errors, but SEMINAL is unique in that it does not look at the types assigned to expressions by the type-checker. Our work also resembles slicing systems because it computes the set of interesting nodes to examine, but is unique in that it does so without any analytical techniques.

The most similar reparation system is from McAdam [16]. He extends the type checker with a grammar of "linear morphisms" to resolve type errors. A morphism from $(\tau_1 * \tau_2) \rightarrow \tau_3$ to $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, for example, can make a currying error type-check. The system can consult what morphisms were generated to produce an error message. (However, this process involves partial-evaluation technology and may not adapt well to other languages.) In addition to working at the type-level, this system also does not search for interesting locations: it is constrained to trying morphisms at the program location where type-inference failed. That is, McAdam's work can present a repair only where the unchanged type-checker would have produced an error message. On the plus side, McAdam's framework can automatically compose morphisms to discover a sequence of changes whereas we can create sequences only via explicit dependent modifications.

Other approaches attempt program analyses (such as slicing) or changes to the type-checker's unification algorithm to determine all the locations that contribute to a program being ill-typed. Prior work [21, 6, 3, 7, 8, 19, 10] indicates that this is a feasible solution to a sound reporting of erroneous code. However, our work dodges many of the difficulties in a proper program analysis, relying instead on the type checker to throw out any invalid suggestions on our part. We also have the ability to make changes rather than just identifying the cause of an inconsistency.

9. Conclusions

We have attacked the ML type-inference error-message problem with an approach that circumvents all the type-inference issues. By finding similar programs that type-check, we have a system that presents a new flavor of error message without changing the type-checker, adding to the trusted computing base, or slowing down compilation of well-typed programs. It is easy to augment our approach with new "changes to try" or new ranking functions for ordering successful changes. Our approach is also robust to changes to the underlying type system.

More speculatively, we believe our approach may improve programmer productivity and pedagogy in ML and other languages with type inference. Fixing type errors by changing the source program matches well how programmers think about their program, so our approach to error messages helps automate what was previously a human-intensive task.

Acknowledgments

Alden King helped identify common Caml typing errors and developed a pretty-printer for our error messages. Marius Nita provided helpful feedback to improve this paper.

References

[1] David T. Barnard, Gwen Clarke, and Nicolas Duncan. Tree-to-tree correction for document trees. Technical Report 95–372, Department of Computing and Information Science, Queen's University, Kingston, January 1995.

[2] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4):17–30, 1993.

[3] Karen Bernstein and Eugene Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.

[4] Philip Bille. Tree edit distance, alignment distance and inclusion. Technical report, IT University of Copenhagen, April 2003.

[5] Olaf Chitil, Frank Huch, and Axel Simon. Typeview: a tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *12th International Workshop on Implementation of Functional Languages*, Aachner Informatik-Berichte, pages 63–69, 2000.

[6] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical Report 426, Indiana University, December 1994.

[7] Dominic Duggan. Correct type explanation. In *ACM Workshop on ML*, pages 49–58, 1998.

[8] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, July 1996.

[9] Milind Gandhe, G. Venkatesh, and Amitabha Sanyal. Correcting errors in the Curry system. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 347–358, London, UK, 1996. Springer-Verlag.

[10] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[11] Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[12] Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *13th ACM Symposium on Principles of Programming Languages*, pages 44–57, St. Petersburg Beach, Florida, January 1986.

[13] Yang Jun, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

[14] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[15] Bruce J. McAdam. On the unification of substitutions in type inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98)*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.

[16] Bruce J. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, Laboratory for Foundatations of Computer Science, The University of Edinburgh, 2001.

[17] B. J. Oommen, K. Zhang, and W. Lee. Numerical similarity and dissimilarity measures between two trees. *IEEE Transactions on Computers*, 45(12):1426–1434, 1996.

[18] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in Haskell. In *ACM Workshop on Haskell*, pages 72–83, Uppsala, Sweden, 2003.

[19] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, 2001.

[20] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[21] Mitchell Wand. Finding the source of type errors. In *13th ACM Symposium on Principles of Programming Languages*, pages 38–43, St. Petersburg Beach, Florida, January 1986.

Appendix

Figure 7 shows the entire recursive search procedure. It calls `enumerate` to retrieve enumerated changes and children on which to recur. For each modification, it uses `doesItTypecheck` to determine if the modification succeeds. It maintains a global context and a local context-window as described in Section 4. It returns a list of modifications that is passed to the ranker.

```
(* types astThing, astRepl, and astMod defined in previous figures *)
val doesItTypeCheck : astThing -> bool
val shiftWindow : (astThing->astThing) array -> (astThing->astThing) -> (astThing->astThing) array
val widestContext : (astThing->astThing) array -> astThing
val enumerate : astThing -> astMod list
```

```
let rec search (wholeCtxt : astThing->astThing)
               (ctxtWindow : (astThing->astThing) array)
               (astNode : astThing) =
  let trySingleRepl (r : astRepl) = doesItTypeCheck (wholeCtxt (r.revision ())) in
  let rec tryMod md =
    match md with
      Recur (childNode,ctx) -> search (fun x -> wholeCtxt (ctx x)) (shiftWindow ctxtWindow ctx) childNode
    | Replace (replacements, followups) ->
        let successes = List.filter trySingleRepl replacements in
        if successes = []
        then []
        else
          let validMods       = List.filter (fun r -> r.validate ()) successes in
          let validWithCtxts  = List.map (fun r -> (r, widestContext ctxtWindow, wholeCtxt)) validMods in
          let followupResults = List.flatten (List.map tryMod (followups ())) in
          List.append validWithCtxts followupResults
  in
  List.flatten (List.map tryMod (enumerate astNode))
```

Figure 7. The SEMINAL search engine