Semantic Navigation of Large Code Bases in Higher-Order, Dynamically Typed Languages

S. Alexander Spoon and Olin Shivers
College of Computing
Georgia Institute of Technology
Atlanta, GA 30032

Abstract— **Chuck is a new code browser that allows navigation of a code base along *semantic* structures, such as data-flow and higher-order control-flow relationships. Employing the fast DDP type inferencer, it is effective on dynamically typed code bases an order of magnitude larger than the code bases supported by previous browsers. Chuck supports the full Smalltalk language, and is now shipped as a standard component of the Squeak open-source Smalltalk system, where it routinely works with code bases exceeding 300,000 lines of code. Chuck's implementation is tuned for interactive use, and is transparently integrated with the Squeak system's existing code-browsing tools. Thus, it provides semantic navigation of a live code base that is still being edited without requiring long pauses for reanalysis due to edits of the code.**

I. Introduction

Higher-order, dynamically typed programming languages occupy a vital and thriving niche in the programming ecosystem. According to its web page, the `ezboard.com` forum-hosting service, which is written in Smalltalk, currently supports over ten million subscribers [1]. The Chrysler C3 software, also written in Smalltalk, includes 2000 classes and 30,000 methods [2]. Over a thirteen-month period in 2002–2003, Cincom reported 1800 new registrations per month for the non-commercial version of their VisualWorks Smalltalk system [3]. It is important to find ways to maintain and reuse these code bases. Since they often have multi-year life spans, and typically include code whose development goes back multiple decades, reverse engineering is important for these systems.

Code browsing tools help reverse engineering by providing reliable information about the code base and by providing navigation paths through the code base. Instead of inferring information through conventions, idioms, textual searches, or possibly erroneous or misleading documentation, programmers using a code browser get reliable information and navigation paths that are determined automatically from the code base according to the syntax and semantics of the language.

Syntactic navigation allows the user to navigate through the code along paths determined by simple syntax-level relations. For example, it allows programmers to navigate from classes to their superclasses and subclasses, from variables to the expressions that use them, and from methods to the methods they override. The standard Smalltalk browsing tools include extensive support for syntactic navigation. *Semantic navigation*, the topic of this paper, allows the programmer to follow paths formed by deeper semantic connections between elements of the code, such as the data- and control-flow induced by the dynamic semantics of the language. (In particular, note that control flow in a higher-order functional or object-oriented language is no longer directly encoded in the syntax of the language.) For example, semantic navigation allows programmers to navigate from a `new` expression to the expressions that use objects created by that expression. It allows programmers to navigate from a message-send expression to the narrow set of methods that might respond to that specific invocation. These links, whose increased precision and tighter bounds help guide and focus programmers' exploration of large code bases, are provided by program analysis.

Powerful and precise semantically-directed code browsers are critically enabling tools for working with large code bases. Without them, code bases can become self-limiting as they grow. Clearly, it helps programmers develop applications when they have access to a large, underlying base of existing code—the programmer can reuse existing components instead of having to reinvent everything from scratch, exploiting the existing investment in design, construction and debugging that the existing code represents. However, when the size of the existing code becomes too large, the programmer can suffer information overload, and may, in the end, decide that the pain of writing the application completely from scratch is preferable to the pain of wandering lost through a tractless wasteland of undifferentiated software, sifting for possibly relevant components to use. Sophisticated navigation aids help programmers, in the popular phrase, stand on each other's shoulders, rather than each other's toes.

Unfortunately, the program analyses required to provide precise semantic navigation for higher-order, dynamically typed languages are polynomial time (or worse) , and so have not scaled to code bases with hundreds of thousands of lines of code. This paper discusses Chuck, a new code browser with semantic navigation for Smalltalk, that *does* scale to large code bases.

Chuck's core technology is the **DDP** type inferencer [4], which provides the necessary static analysis. However, in the process of building Chuck, we also learned that such a tool faces challenges beyond simple asymptotic scalability. We had to design the system to produce fast results *consistently*, not just on average. Further, Chuck is required to analyze live code bases where, as dynamic-language practitioners expect, the programmer can tightly interleave code browsing and code editing. Such environments require reanalysis whenever the code is edited but must not allow that reanalysis to pause overly long and induce programmer impatience.

Chuck is implemented in and for Squeak [5], a production-level implementation of Smalltalk [6], and is now a load option in the standard releases of Squeak. Chuck is hosted on SqueakSource[1], and at the time of this writing has the second-largest number of downloads of the 200+ Squeak projects hosted on that site[2]. We infer from this statistic that the tool addresses a true need of Squeak programmers.

After reviewing **DDP**, this paper describes the four navigation paths provided by Chuck and the solutions Chuck has to the two other challenges mentioned above. We then connect this research to several areas of related work, describe future work on Chuck, and conclude.

As a note on terminology, when we say "large code base," we consistently mean code bases with hundreds of thousands of lines of code.

II. Type Inference with DDP

The type-inference problem is similar for various dynamic languages, including Smalltalk [6], Self [7], Scheme [8], and Cecil [9]. Algorithms that are effective in one language tend to be effective in the others. All of these languages share the difficulty that the analyses of control flow and data flow are interdependent [10]; all of these languages have and use dynamic dispatch on types and have data-dependent control flow induced by objects or by higher-order functions.

Efforts on this type-inference problem date back at least two decades. Suzuki's work in 1981 is the oldest published work that infers types in Smalltalk without any type declarations [11]. More recently, in 1991, Shivers formalized *context*, using *abstract contours*, as a central technique and a central design difference among type-inference algorithms [12]. Later, the core of Agesen's "Cartesian Products Algorithm" (**CPA**) is the insight of choosing contexts based on data-flow information: **CPA** selects contexts as tuples of parameter types [13]. (The algorithm gets its name because the tuples are chosen as cartesian products of the possible argument types for each method.) More recently yet, Flanagan and Felleisen have increased the speed and scalability of type-inference algorithms by isolating part of the algorithm to work on individual modules. By doing so, their algorithm spends less time analyzing the entire code base [14].

Unfortunately, the existing algorithms do not scale to large dynamically typed code bases. Certainly, no successful results have been reported for such large code bases. Grove, *et al.*, implemented a wide selection of these algorithms as part of the Vortex project, and they report that all of the context-sensitive algorithms require an unreasonable amount of time and memory for larger Cecil programs [15]. They write:

> Larger Cecil programs may benefit from context-sensitivity, but our experiments demonstrated that scalability problems prevent the context-sensitive algorithms from being applied beyond the domain of small benchmark programs. Thus the search for a scalable and effective call graph construction algorithm for programs that make heavy use of polymorphism and dynamic dispatching remains an open problem.

There are linear-time unification-based algorithms that have great promise [16], but they have yet to be proven practical for the present application. First, these context-insensitive algorithms sacrifice precision for speed; it is not clear if they would produce usefully precise results on large, extremely polymorphic, higher-order code bases. Second, it is difficult to use even a linear-time algorithm for semantic browsers that allow interleaved code changes. The linear-time algorithms are linear *in the amount of code* and thus require a substantial amount of computation for large code bases.

DDP is a recently developed type-inference algorithm which is effective for Smalltalk code bases with hundreds of thousands of lines of code. The details of the algorithm are published separately [4]. This section gives an overview from the perspective of a client of the algorithm.

The essence of **DDP**'s approach is to cast the problem as a demand-driven, backwards-chaining algorithm that *prunes* goals. It posts goals to itself, such as "What is the type of variable `x`?," and then tries to solve those goals. Solving a goal usually requires posting other goals. For example, finding the type of a variable requires finding the type of each expression ever assigned to that variable. Whenever the algorithm needs a goal that is not already being pursed, it creates a new one and adds it to the *goal pool* of goals being pursued.

The kinds of goals **DDP** can pose to itself correspond closely to the four navigation paths that Chuck allows. Thus, a discussion of the kinds of goals is deferred until Sec. IV.

All demand-driven algorithms have the structure described so far [17]. What sets **DDP** apart from other demand-driven algorithms is that it sometimes *prunes* goals instead of finding a precise answer for them. Pruning a goal means that the goal is given a sufficiently conservative solution that the goal no longer requires other goals to find its answer. An example would be "`x` is of type *Anything*." Pruning provides a technique for the algorithm to control how hard it is working. It is a technique for sacrificing precision to gain scalability. The heuristic implicit in using pruning is the notion that conservative approximations that occur far from the original query in the subgoal tree don't hurt precision too much—analogous to the similar tactic a chess program might use of pushing approximate board evaluations deep into the search tree.

The standard **DDP** pruning algorithm chooses prunings in order to limit the total number of goals being studied. The

[1] `http://www.squeaksource.com`

[2] Chuck has 3698 downloads while Seaside has 4141. These are raw download counts that not been processed or filtered in any way, and thus a more careful analysis might adjust Chuck's relative ranking by a few levels. The point would remain, however, that Chuck is among the most-downloaded projects implemented in Squeak.

standard algorithm is parameterized by a *pruning threshold*, K. Whenever the analyzer has created K new goals, it prunes enough goals that only K goals remain relevant to the main goal. After a pruning, any goals that no longer effect the main goal, either directly or indirectly, are removed from the goal pool. Thus, once **DDP** has created its first K goals, the goal pool varies between K and $2 * K$ goals for the remainder of the analysis.

Instead of requiring time that is linear or more in the size of the code base, **DDP** appears, according to experiments to date, to require time proportional to the pruning threshold. This result is intuitive for the relatively small choices of pruning threshold used in practice. Each goal tends to be updated only a very few times, and with a small threshold, the algorithm only manipulates small types and other data-flow structures. This performance characterization is promising for use in interactive tools that operate on large code bases.

Chuck modifies this pruning algorithm as described in Sec. III in order to support interactive use.

The performance of **DDP** has been measured empirically [18] in an extension of a previously reported experiment [4]. While the details of these experiments are beyond the scope of this paper, the results bear summarizing because they influence the tuning of **DDP**'s pruning algorithm for use in Chuck.

In our experiments, a typical Squeak code base was assembled based on Squeak 3.7. The resulting code base, henceforth called "the experimental code base," has 358,872 non-blank lines of code, 2485 classes, and 48,715 methods. We selected nine applications within this code base and used **DDP** to infer types for the instance variables declared in those applications. Each variable was the target of 12 type-inference queries, where each query used a different pruning threshold ranging from 50–10,000. Each query was timed, and each inferred type was classified as precise or imprecise. **DDP**'s performance under each pruning threshold can then be summarized with a pair of statistics: the average time required and the percentage of results that are designated precise.

These results suggest that 3000 is a moderate, balanced choice of pruning threshold. With a pruning threshold of 3000, **DDP** answers type queries in an average of 30 seconds, and of those answers, 49% are precise. On the other hand, an extreme choice of threshold is 50. With a pruning threshold of 50, the algorithm answers type queries in an average of less than one second, but still answers 35% of the queries precisely.

III. Pruning Algorithm for Interactive Tools

DDP can be tuned to trade off between speed of analysis and quality of results. It does so by tuning its *pruning algorithm*. Chuck modifies **DDP** to use a pruning algorithm specifically suited for use in interactive tools.

The standard **DDP** pruning algorithm limits the number of active goals according to a fixed *pruning threshold* chosen before the analyzer begins. As described earlier, choosing a lower pruning threshold tends to increase the speed of the inference but decrease the final answer's precision. This behavior is only a tendency, however, and individual queries have widely varying performance. As described in the previous section, a threshold of 3000 nodes yielded an average time of 30 seconds per query, but the slowest of those queries required over 10 minutes.

For interactive use, these occasional large response times are not acceptable. We would prefer to provide consistently fast responses even if the responses are not as precise as possible. A crude way to obtain consistently fast responses would be simply to halt the algorithm if a response has not been found within some time limit and report failure. That is, run the analysis, and if it requires more than, say, five seconds, terminate it and report that no information was found.

Chuck obtains a more graceful degradation of precision than this drop-dead approach by taking advantage of the structure of the **DDP** pruning algorithm. The tool begins by using the standard pruning algorithm with a pruning threshold of 3000. If no result has been found within three seconds, then the pruning threshold is decreased to 50 and the algorithm is given two more seconds to complete. Usually the algorithm finishes in a fraction of a second with a pruning threshold of 50, but in the unusual case that it requires two or more seconds, the algorithm is terminated after all and the system reports that no information was found.

Based on the experiments we've performed, the crude approach of using a threshold of 3000, and stopping after five seconds yields an answer—precise or imprecise—to 40% of type queries. The remaining 60% would necessarily have to be answered with the maximal type, *Anything*. The more gradual approach finds answers to a total of 94% of the queries: it answers 37% in the first three seconds, and 57% after reducing the threshold to 50. Both approaches have a maximum execution time of five seconds, but gradual reduction of the threshold yields a complete analysis of many more queries. Additionally, the gradual-reduction approach has an improved expected analysis time of 2.6 seconds instead of 3.3 seconds.

With this pruning algorithm, the time required per query does not depend on the speed or load of the underlying machine. All queries finish in five seconds. Instead, the speed and load on the underlying computer affect the *quality* of results that **DDP** produces. A slower machine will still finish each query in five seconds but will produce less precise results.

IV. Semantic Navigation Paths

Chuck uses program analysis to provide navigation along four paths in the semantic structure of the code base:

- A *flow query* asks where the value of a computation could flow.
- A *type query* asks what kinds of values could flow to a given expression.
- A *responders query* asks where control could go at a given method invocation.
- A *senders query* asks which program points could transfer control to a given method.

Each navigation path is useful to programmers at different times. The different paths also exactly correspond to the kinds of goals that direct search internally in the DDP analysis.

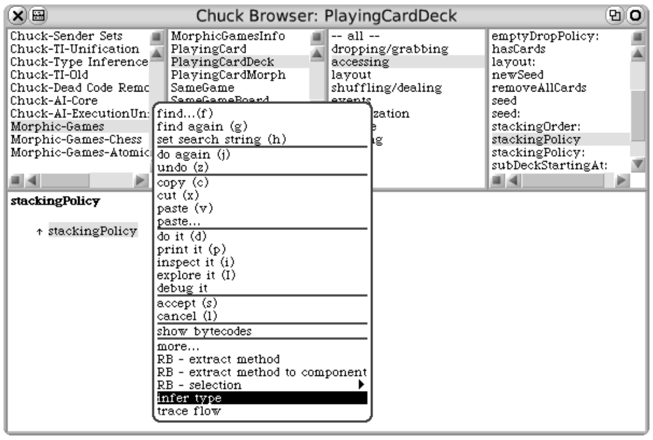

Fig. 1. Chuck is seamlessly integrated with the standard Smalltalk development environment. In this screenshot, the programmer is exploring code using the Squeak system's basic code browser, and requests the type of a variable reference occurring in the source code. The top four panes, from right to left, show that the programmer is examining (1) method `stackingPolicy` of (2) the `accessing` group of (3) class `PlayingCardDeck`, which is in (4) the `Morphic-Games` application. The pane occupying the bottom half of the screen shows the code for the method, a simple instance-variable fetch. The programmer has highlighted the variable `stackingPolicy` and, with a mouse click, called up the standard Squeak context menu, which now includes the Chuck operations relevant to the selected code, "infer type" and "trace flow." The "infer type" operation has been selected. The result of the query, with the response's supporting derivation, is shown below, in Fig. 2

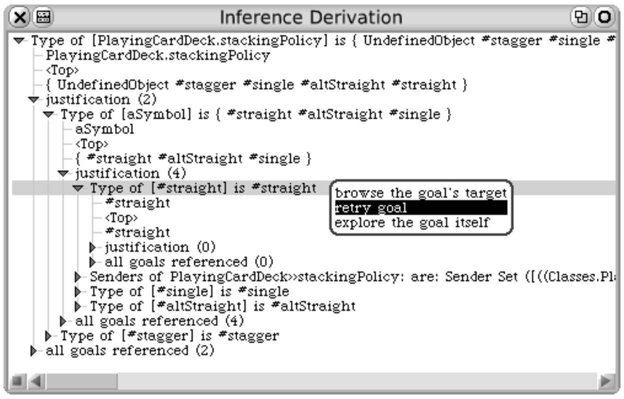

Fig. 2. Chuck answers the query from Fig. 1 with a *derivation browser*. The top line of the browser shows the initial query: "What is the type of `stackingPolicy` in class `PlayingCardDeck`?" The derivation shows a tree browser, and the children lines of the top line show details about the initial query. The first two lines describe the query in detail, while the fourth gives the inferred type. The next two children lines show the subsidiary queries used to find a type for the initial query. These children lines can be expanded in turn to show details about the associated queries. In this screenshot, the user has expanded multiple queries and is about to ask Chuck to "retry goal," that is, try the selected query again but with a higher pruning threshold.

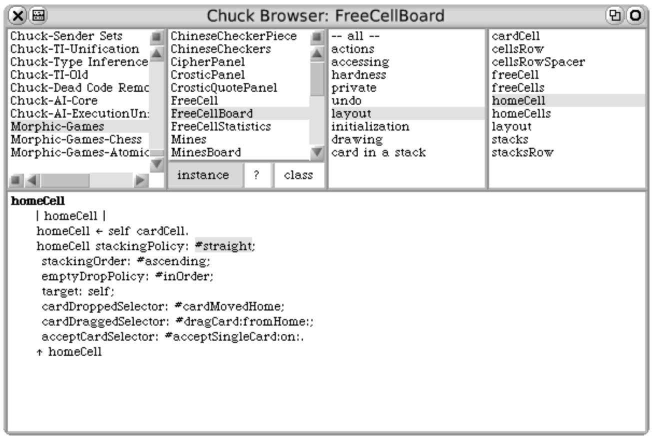

Fig. 3. This screenshot shows the user returning to the browser after tracing through the derivation browser (shown in Fig. 2) to an interesting piece of code. The highlighted code, a literal for `#straight`, is the code the user had selected in the derivation browser before returning to the code browser.

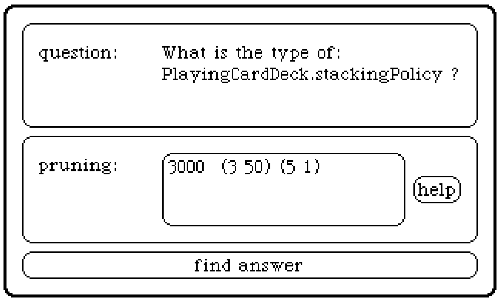

Fig. 4. If a user asks to retry a goal, this dialogue appears, allowing the user to change the pruning schedule and thus retry the goal with more effort.

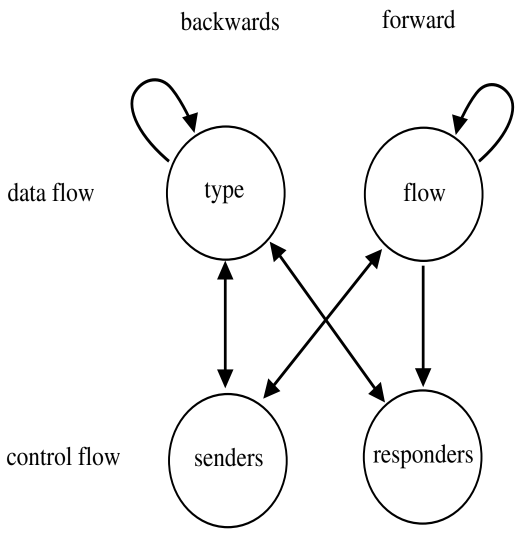

Fig. 5. The four queries Chuck can answer along with their dependencies on each other.

Flow queries

A *flow query* asks where the value in a variable or expression will flow when code in the code base runs. This information is useful to see where something is ultimately used. For example, a user can select the variable `RadiansPerDegree` in class `Float`, ask where the variable flows, and be told that the variable flows to the methods `radiansToDegrees`, `degreesToRadians`, `*`, and `/` in class `Float`. By reading the code of these methods, the user can see that `radiansToDegrees` and `degreesToRadians` refer to the variable in order to convert angles between radians and degrees, and in turn those two methods pass the number to the `*` and `/` methods.

The available locations that Chuck reports in response to a flow query are precisely the flow positions of **DDP**. The following flow positions are possible:

- **Variables**
  For example, `Display` is a flow position designating the values assigned to the `Display` global variable during program execution.
- **Expressions**
  Any expression is a flow position designating the values the expression might produce at run time.
- **Methods**
  For example, method `next` of class `Random` is a flow position designating values held by the receiver (`self`) of the specified method.
- **Sets of the above**
  Any set of the above flow positions is itself a flow position.
- **Anywhere**
  The flow position *Anywhere* describes all flow positions in the entire code base.
- **Nowhere**
  The flow position *Nowhere* is a flow position that never holds a value. If a variable is never used, then it flows to *Nowhere*.

These flow positions are additionally discriminated by static contexts provided by means of the analysis' abstract contours; this is a level of detail beyond the scope of this paper [18].

Type queries

A *type query* asks what kind of values a variable or expression will hold when the code base runs. For example, the programmer might ask what type of values is held by the `stackingPolicy` instance variable of class `PlayingCardDeck`, as shown in Fig. 1. Perhaps the programmer is considering using this class but wonders what options are available for this setting.

The tool would respond in this case that the variable might hold either the value `nil` or one of the symbols `#stagger`, `#single`, `#altStraight`, or `#straight`, as shown in Fig. 2. The programmer thus learns that the variable holds an *ad hoc* enumerated type. The tool would then allow the user to navigate backwards along the data-flow paths that gave rise to each option. Continuing the example, the user could select the symbol `#stagger` and then navigate to the code which caused Chuck to include it as a possible type.

The types returned are exactly the types derived by **DDP**. The possible types are:

- **Individual class**
  For example, `PlayingCardDeck` is a valid type which includes all instances of that class.
- **Individual symbol**
  For example, `#straight` is a type which includes only the symbol object named `straight`.
  It's worth asking why we bother to build this kind of precision into the static abstractions used by Chuck's analysis—this is, after all, a design decision in the construction of the tool. There are two idioms or patterns of use in Smalltalk that drive the need for this precision. First, as we have seen in our playing-card example shown in the figures, sets of symbols are frequently employed by Smalltalk programmers to serve as *ad hoc* "enumerated types." Thus it's important to be able to track these symbols through the program's data flow in order to produce type reports that correspond to the programmer's internal model.
  A different pattern employing symbols is critical for determining program control flow. Smalltalk programmers frequently use symbol-driven key dispatch in the event loops of application GUIs. The dispatch is implemented by means of the `perform:` method, which allows the message sent to an object to be specified as a data value (a symbol), rather than a fixed name. Chuck's DDP analysis correctly traces method invocation that is dispatched by `perform:` indirection, given the reasonable restriction that symbols used in `perform:` dispatch must be literals appearing somewhere in the program (rather than data computed from, *e.g.*, strings).
  Thus determining the control flow of a GUI's event-handling loop requires determining which symbols flow to the handler's `perform:` dispatch. Without this level of discrimination in the analysis' static abstractions, the static control flow of this kind of code would collapse into an unworkably conservative "crossbar" of possible control edges, which would, in turn, poison the precision of any data flow connected to the event-handling code.
- **Individual block**
  A block type specifies a particular block from the source code. Smalltalk blocks are akin to Scheme lambda expressions. A block type includes all closure objects which were created by evaluating the specified block. Note that handling blocks with precision is critical for semantic analysis of Smalltalk programs. Smalltalk blocks are first-class values used in a pervasive and fine-grained way. For example, the basic if/then/else construct in Smalltalk is provided by sending two block objects to the boolean selector. Failure to handle blocks in a polyvariant manner would confuse together the control flow of every single conditional branch in the entire code base.

- **A set of the above**
  Any set of types of the above kinds is itself a type. For example, "`#straight` or `PlayingCardDeck`" is a type including the symbol `#straight` and all instances of class `PlayingCardDeck`.
- **Anything**
  Type *Anything* includes every possible object.
- **Nothing**
  Type *Nothing* includes no objects at all. Occasionally a variable is never assigned a value (for example, the parameters of a method that is never invoked).

Again, types are additionally discriminated by abstract-contour context, which is particularly important for higher-order values such as blocks.

Responders queries

A *responders query* asks what methods might respond when a particular message-send expression executes. As an extreme example, if one browses to class `BasicLintRuleTest`'s `new` method in the experimental code base, and selects the message send of `initialize`, the standard syntax-directed query shows 756 potential responders. Chuck's enhanced semantic query uses type information and shows only one.

Senders queries

A *senders query* asks what expressions might invoke a method. This is a useful tool for code exploration. For example, a programmer might wonder what is the standard way to use class `HtmlParser`. Seeking examples, the programmer might ask what expressions invoke the class's `parse:` method; in the standard Squeak code base, Chuck will show the exact six expressions that can possibly invoke the method at run time.

Chuck's senders query enhances the standard Smalltalk senders query. The standard query is syntactic: it only looks at the name of the method invoked by a send statement. In our `HtmlParser` example, it will return ten expressions including irrelevant false-positive invocations, including code that invokes VRML and email parsers. More extreme examples exist: methods with common names such as `initialize` often have only one sender according to Chuck, but hundreds of senders according to the standard tool.

Queries and semantic structure

Chuck's four navigation paths correspond to flow queries that vary on two axes. Fig. 5 shows this diagrammatically. The two axes are *control- versus data-flow* and *direction of flow*.

A *control-flow* question asks about the links in the control structure of the program. In a higher-order language, these paths are determined by the dynamic semantics; they are not simple syntactic links. Responders queries and senders queries are both control-flow questions: they ask about the order of execution of message-send expressions and the class methods they invoke.

A *data-flow* question, to contrast, asks about the paths through which data can flow at run time. A flow query asks for the locations to which a value can flow, given a specified start location. A type query finds the type of an expression by finding the locations that contribute a value that will flow to the expression at run time.

The *direction of flow* of a query can be *forward* or *backwards*. A *forward* query asks what will happen in the future as the code base executes. Responders queries and flow queries are both forward queries. A responders query asks for the methods to which control might transfer in response to a message-send expression, whereas a flow query asks what expressions might later hold a value that is currently stored in a specified start expression.

A *backwards* query asks about the past of program execution. A senders query is a backwards query because it asks which message-send statements in the past caused control to transfer to the specified method. A type query asks, in the course of finding its final answer, which expressions in the past can flow to a specified expression.

V. Explanations and Trying Harder

In addition to providing navigation paths and raw information, Chuck *explains* its answers using a *derivation browser*. It explains the answer to each query by showing the other queries used to support that answer. Those queries in turn have their own explanation, leading naturally to the expandable tree view shown in Fig. 2. Each query shown in the derivation browser may be expanded to see information about the query. When a query is expanded, the derivation browser shows the question that query asks, the answer **DDP** found to the question, and the subsidiary queries used to find that answer. The user may expand these subsidiary queries, and then expand their queries, *etc.* By doing so, the user simultaneously traces through an explanation of the original query's answer and traces through semantic paths of the code base.

Consider the example shown in Fig. 2. A programmer has asked about the type of variable `stackingPolicy`, and Chuck has reported that the type is a set of symbols. In order to learn why `#straight` is included in the set, the user can trace one level down and see that `aSymbol`, the first parameter of method `stackingPolicy:`, can hold `#straight`. Tracing through that claim's explanation, the user sees that some method is invoking `stackingPolicy:` with `#straight` passed as a literal argument. Finally, the derivation browser also allows a programmer to retry a goal with more effort. **DDP** has a fundamental trade off between time required and precision of results, and sometimes it finds an imprecise result simply because it returned an answer too quickly. By default, **DDP** uses the pruning schedule described in Sec. III and thus reliably finishes within five seconds. If a programmer is unsatisfied with the answer and is strongly interested in the particular question, then the user may use a context menu in the derivation browser and ask the tool to try the query again with more effort expended.

The *try harder* dialog is shown in Fig. 4 with the default level of effort depicted: the algorithm begins with a pruning threshold of 3000, decreases to 50 after 3 seconds, and decreases to 1 after a total of 5 seconds. To find better results at the expense of more time, a user may enter a different pruning schedule—for example, increasing the 3 seconds to 30 seconds—and then select "find answer."

VI. Integration with Standard Tools

It's worth stressing that, although Chuck is the product of the first author's doctoral research, it is not a "research prototype." It works on full-blown Smalltalk and is now a standard component shipped with the Squeak open-source programming environment for Smalltalk.

Chuck integrates seamlessly with the standard Squeak programming tools. Users do not execute a visibly separate tool in order to use Chuck's additional queries. Instead, they use ubiquitous context menus to navigate between Chuck windows and regular browser windows.

Users first invoke Chuck from the regular code browsers. They select something of interest, such as an expression, variable, or class, and then invoke the context menu using a designated mouse button. This interface is familiar to users because it is already used for refactorings and the standard (syntactic) navigation paths [19].

The menu items added depend on the item selected. If the user selects an expression or variable, then the menu includes options for type queries and flow queries. If the user selects a message-send expression, the menu additionally includes a responders query. For a method, the menu includes a senders query.

Users can also return to existing Squeak tools from Chuck. Again, context menus provide the interface. If a user selects any expression, method, or variable in a Chuck derivation browser, then the context menu allows the user to "browse it" and return to a regular Squeak code browser displaying the selected item.

Continuing the example from the previous section (see Fig. 2), the user might wonder what method is passing `#straight` as an argument to `stackingPolicy:`. The user could select "browse the goal's target" and return to a code browser pointed at method `homeCell` of class `FreeCellBoard`, as shown in Fig. 3. Since `FreeCellBoard` is part of the Free Cell game implemented in the code base, and `homeCell` is presumably a method to create a "home cell" area of a Free Cell board, the user now has enough information to locate an example use of `PlayingCardDeck` that uses the `#straight` stacking policy: the user can start a game of Free Cell and then inspect the stack of cards in one of the home cells.

VII. Analyzing Live Code

Chuck operates on live code, just like the programming tools of Self [7], Dr. Scheme [20] and Smalltalk. Users do *not* need to run the analyzer overnight and then browse the results the next day. Instead, users may intermingle code browsing and normal code editing, and Chuck will recompute data-flow information as needed.

To accomplish this, the tool is sufficiently integrated with the standard Squeak tools that it can detect when the code base changes and update its internal information accordingly. The processing is correspondingly divided into two phases: semantic analysis that is performed at the time of the query, and syntactic analysis that is performed in advance and is updated each time the programmer commits an edit, e.g., each time the programmer defines a new class.

The syntactic information maintained by Chuck consists of the following tables:

- `parseTree`, which maps each method to its parse tree.
- `methodsImplementing`, which maps each possible method name to the list of methods in the system that implement the method. This is used in responders queries when the receiver type is *Anything*.
- `expressionsSending`, which maps each method name to the list of `send` statements which send the name. This is used to find the `send` statements that may invoke a method.
- `symbolLiterals`, which maps each method name to the list of literal expressions that mention a symbol for that method name. This table is used for analyzing Smalltalk's `perform:` method, a method that invokes methods by a computed name.
- `assignmentsDefining`, which maps each variable to the list of statements which assign something to that variable. This is used when processing type queries.
- `expressionsReading`, which maps each variable to the list of statements that read from that variable. This is used when processing flow queries.

This information requires approximately 65 megabytes of memory to maintain for a typical development image (the standard Squeak 3.7/Basic release with Chuck loaded, about 306,000 lines of code).

The updates required after code changes are straightforward and proportional to the size of the code being altered. For example, if a new method is added to the code base, then:

- The method's parse tree is added to `parseTrees`.
- The method is added to the list in `methodsImplementing` corresponding to the method's name.
- For each message-send expression in the method, the expression is added to the appropriate list in `expressionsSending`.
- For each literal expression in the method where the literal is a symbol, the expression is added to the symbol's list in `symbolLiterals`.
- For each assignment statement in the method, the statement is added to list of statements in `assignmentsDefining` corresponding to the variable on the left-hand side of the assignment statement.
- For each variable expression, the expression is added to

the appropriate list in `expressionsReading`.

These changes are reversed when a method is removed. All other code changes are described as a combination of method removals followed by method additions. Examples of such changes are:

- If a method is changed, then the tables are updated as if the method was removed followed by the method being added.
- If a class definition is changed, then the tables are updated as if every method in the class and its subclasses were removed before the definition change and added back after the definition change.

The time taken for these hash-table operations are swamped by the overhead of the Squeak system's dynamic Smalltalk byte-code compiler, which is invoked in response to code changes. For example, when a method is added, Chuck adds entries to its tables for that method, and the system compiles the method to byte code. The additional overhead involved in maintaining the syntactic data structures is imperceptible.

VIII. Related Work

Chuck adds to a number of ongoing threads of research. This section describes Chuck's relation to significant work from each of these threads.

A. Navigating with Inferred Types

Chuck most closely relates to the general topic of navigation with inferred data-flow information. Research in this area uses static analysis in order to improve code-browsing tools.

For example, Agesen and Madsen built a type-based browser for Self based on the **CPA** type-inference algorithm [21]. This type-based browser uses data-flow information inferred in a whole-program analysis. The browser uses this information to display additional type information. Further, the browser can browse methods in the context of particular argument types, *e.g.* to browse the `*` method of class `Float` when the receiver and the argument are both `Float`'s. With the approach of this browser, if the program changes then currently open browsers become invalid and the analysis needs to be repeated.

Another example is Mr. Spidey, a semantic browser that integrates with the PLT dialect of Scheme [22], [23]. Mr. Spidey includes similar navigation and information paths as Chuck and as Agesen and Madsen's type-based browser.

Both of these example systems make some effort at type checking in addition to type-based browsing. If an expression's type indicates that it might cause a type error at run time, then the expression is colored red. We do not have any data establishing whether this feature is effective, and not merely possible, using types inferred with DDP. We do not expect the DDP-inferred types to be sufficient by themselves for effective static type checking, and thus Chuck does not include type-checking feedback. The approach might become more promising for large Smalltalk code bases if type inference and soft types [24] become a common part of Smalltalk development culture and thus programmers frequently try to remove falsely marked errors in their code before releasing it.

Chuck has three contributions over existing tools of this family. First, it is effective in larger code bases than have been demonstrated with existing tools: Chuck is effective on code bases with hundreds of thousands of lines of code. Second, Chuck queries may be freely interleaved with edits to the code base; programmers do not have to worry about when to re-execute the type inferencer in light of new code edits. Third, its analyzer can effectively be tuned for interactive use without simply imposing a drop-dead time limit.

B. Type Inference for Program Understanding

Data-flow analysis has been studied for other program-understanding applications besides improved code browsers. O'Callahan and Jackson have written a type-inference tool for C called Lackwit [25], and Deursen and Moonen have written a similar tool for Cobol [26]. In both cases, the authors argue that type inference in some suitably enriched refinement of the base language's type system can provide useful information for program understanding even though the languages in both cases are already statically typed.

Chuck shares the spirit of such work but studies type inference in the specific context of semantic browsers for large dynamic code bases.

C. Slicing

Slicing tools find the parts of a program that are relevant in various ways to a user-selected portion of the program [27]. For example, they can find the portion of a program necessary to cause a selected variable to gain its first value. There are many variations on the basic theme of slicing. The slicer might choose enough of the program to give a variable all of its values instead of just the first. The slicer might select the portion of the program executing *after* a selected variable. Instead of insisting on a fully executable slice, the slicer might select a smaller portion of code that seems more directly relevant to the programmer's question.

Semantic browsers such as Chuck present similar dependency information in a different way. Instead of immediately displaying the entire portion of a program that is relevant to a selected portion of code, a semantic browser let programmers navigate through the code along data-flow and control-flow paths one step at a time. This approach can help keep a programmer from becoming overwhelmed in cases where the slice is large.

D. Navigating with Unsound Types

Finally, another direction is to use unsound types that do not capture all possible types. Robert, Brant and Johnson's refactoring browser [19] uses dynamic type information to guide its work. Alternatively, the tool can rely on matching common idioms, *e.g.*, extracting information from naming conventions such as "Hungarian notation," which allow programmers to tag variables with extra unchecked type annotations above and beyond what is provided by the language's actual static semantics.

Such approaches do not find fully reliable information, but they still have uses. They can typically run very quickly because they do not need to check for unusual cases or respect the actual language semantics. Further, they can address program-analysis problems for which no analyzer is yet effective. While we confess to a certain unease at the use of unsound analysis systems, we note that program-development tools can provide both false positives as well as false negatives as long as they "do the right thing" often enough for programmers to perceive benefit.

Still, *ceteris paribus*, it's better to have a sound source of information than an unsound one. The present work extends the applicability of tools that use static analysis instead of an unsound source of type information. This work can be viewed as one demonstration that analysis-based tools can plausibly be as effective on large code bases as tools that do not give any soundness guarantee. It would be interesting future work to compare directly the speed and precision of the two kinds of tools.

IX. Future Work

The Chuck browser has been available in the standard Squeak open-source Smalltalk programming environment since July 2004. The tool provides a platform for us to explore technology that supports humans engaged in program understanding.

Chuck is an ongoing development project. We continue to extend its capabilities in various ways. Some directions we are currently pursuing are:

- **Improved inference rules**
  The current rules of **DDP** are complete and conservatively capture the application-writing portion of Smalltalk, but improved rules can give more precise information. One improvement to the rules being pursued is the use of abstract data contours to precisely analyze data polymorphism, *e.g.*, that different instances of the `Set` class hold different types of elements. The inference rules of **DCPA** [28] are helpful but would need updating for Smalltalk's differing idioms. In particular, Smalltalk code bases frequently have only one or two `new` statements in the entire code base, and thus abstract data contours should be based on something like mentions of class names that are later instantiated, instead of `new` statements *per se*.
  A second inference-rule improvement being pursued is to improve senders-of queries for methods with common names like `initialize`. The standard **DDP** rules starts with the hundreds of methods calling `initialize` and then narrows that list using type information. In such a case it should be faster to reverse these steps. The analyzer could start with every message-send expression whose receiver is of an appropriate class, and then narrow that list down by considering which method name each of those expressions sends.
- **Improved pruning algorithm**
  Manual review of Chuck's results suggest that many imprecise results could be greatly improved if the algorithm were to choose prunings more carefully. A particular improvement that is being explored is to change the measure of distance to the root goal that is used to determine the relative priority of different goals. The new measure will consider whether the goal contributes directly to the root goal's result (*e.g.*, when the root goal must have a supertype of the type found by the goal), or is only used indirectly (*e.g.*, when the goal provides call-graph information that will in turn be searched). The pruning algorithm should be extremely reluctant to prune goals that directly influence the root goal, because pruning such a goal produces equivalent results to pruning the root goal itself. This notion of directness appears to generalize beyond direct *vs.* indirect to an integer number of levels of indirection.
- **Off-line processing**
  While Chuck is primarily structured to support interleaved browsing and code editing, we have turned our thoughts to attempting to exploit off-line computation to assist on-line interaction. The scenario is that we assume the computer might have eight hours a night during which the code base is quiescent and not under development. Could the machine run some kind of whole-program, batch analysis over the code base that could assist queries when development resumes each following morning?
  Note that whatever data the machine computes during this off-line period must not be instantly invalidated the moment the programmer makes the first change of the day to the system. To make effective use of off-line, batch analysis, the information computed must remain of utility so long as the code has not changed too much between the time of the analysis and the time of the programmer queries.
  One promising avenue is to cache both results and explanations from the batch analysis. If the user makes a query that was also made in the batch execution, then the cached result can have its explanation checked again to ensure that it is still valid. In many cases the explanation is likely to remain valid because the code has not changed much. Even in cases where the explanation is no longer sufficient, the analysis can begin with the cached information instead of beginning from scratch. There is some risk that the result found in this way is less precise than would have been found with a from-scratch analysis that used the same amount of time. The tool, therefore, might also repeat the five-second analysis, just in case, and provide to the user whichever result is better.
- **Further integration with Squeak**
  Additionally, we plan to integrate Chuck further into the Squeak programming environment by using Robbes' `Services` framework [29]. Currently, Chuck is directly integrated with the code browser; with Robbes' framework, it could also be invoked from other tools that present code to the user, including debuggers, inspectors, and workspaces. Instead of enhancing the browser tool

itself, Chuck would enhance all tools that display code.

- **User Studies**
  Chuck's interface has been tuned in response to feedback from a limited set of users: our coworkers in the college. We plan to run a more structured user study with a larger set of users in order to improve the usability further and in order to test whether Chuck is useful in its intended role.

X. Conclusions

Chuck demonstrates that semantic navigation is practical in large, higher-order, dynamically typed code bases, even when the semantic navigation is interleaved with code edits. To achieve this, Chuck combines a number of techniques:

- Its analyzer uses subgoal pruning in order to get context-sensitive information much faster than algorithms that require time proportional to the size of the code base.
- It uses a pruning algorithm tuned for interactive timings.
- It spends memory to maintain tables about the syntax of the code base as the code changes.

Because the tool tolerates code changes, it can be thoroughly integrated with existing programming tools. Queries can be initiated from the standard code browsers, and when a user traces a query's explained result, the user can reenter the standard code browsers at any point.

Semantic navigation is just one application of type inference. Other tools that can benefit include improved refactorings [19], lint-like tools that find potential errors [22], dead-code removal [21], and optimizing compilation [15]. **DDP** potentially enables a whole class of analysis-based tools for large higher-order, dynamically typed code bases that should be explored.

References

[1] (2001, May) ezboard ranked in top 100 sites worldwide. ezboard.com press release. [Online]. Available: http://www.ezboard.com/corporate/pressroom/2001_03_24.html

[2] A. Radding, "Simplicity, but with control," *Information Week*, vol. 381, April 2001.

[3] J. Robertson. post on november 28, 2003. `comp.lang.smalltalk` USENET group.

[4] S. A. Spoon and O. Shivers, "Demand-driven type inference with subgoal pruning: Trading precision for scalability," in *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[5] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: The story of Squeak, A practical Smalltalk written in itself," in *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1997.

[6] A. C. Kay, "The early history of smalltalk," in *The second ACM SIGPLAN conference on History of programming languages*. ACM Press, 1993, pp. 69–95.

[7] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1987.

[8] I. N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, J. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson, "Revised^5 report on the algorithmic language scheme," *SIGPLAN Notices*, 1998.

[9] C. Chambers, "The cecil language specification and rationale," Department of Computer Science and Engineering, University of Washington, Tech. Rep. TR-93-03-05, March 1993.

[10] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, Carnegie Mellon University, 1991.

[11] N. Suzuki, "Inferring types in smalltalk," in *ACM Symposium on Principles of Programming Languages (POPL)*, 1981, pp. 187–199.

[12] O. Shivers, "The semantics of scheme control-flow analysis," in *Partial Evaluation and Semantic-Based Program Manipulation*, 1991, pp. 190–198.

[13] O. Agesen, "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[14] C. Flanagan and M. Felleisen, "Componential set-based analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 2, pp. 370–416, 1999.

[15] D. Grove, G. Defouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1997.

[16] G. DeFouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1998, pp. 222–236.

[17] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow," in *Symposium on Principles of Programming Languages*, 1995, pp. 37–48.

[18] S. A. Spoon, "Demand-driven type inference with subgoal pruning," Ph.D. dissertation, Georgia Institute of Technology, 2005.

[19] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253–263, 1997.

[20] J. Clements, P. T. Graunke, S. Krishnamurthi, and M. Felleisen, "Little languages and their programming environments," in *Monterey Workshop*, 2001.

[21] O. Agesen, "Concrete type inference: Delivering object-oriented applications," Ph.D. dissertation, Stanford University, 1995.

[22] C. Flanagan and M. Felleisen, "A new way of debugging lisp programs," in *Proceedings of Lisp Users' Group Meeting (LUGM)*, 1998.

[23] C. Flanagan, "Effective static debugging via componential set-based analysis," Ph.D. dissertation, Rice University, 1997.

[24] R. Cartwright and M. Fagan, "Soft typing," in *PLDI*, 1991, pp. 278–292.

[25] R. O'Callahan and D. Jackson, "Lackwit: A program understanding tool based on type inference," in *ICSE '97*, 1997.

[26] A. van Deursen and L. Moonen, "An empirical study into cobol type inferencing," *Science of Computer Programming*, vol. 40, no. 2–3, pp. 189–211, July 2001.

[27] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.

[28] T. Wang and S. F. Smith, "Precise constraint-based type inference for Java," *Lecture Notes in Computer Science*, vol. 2072, pp. 99–117, 2001.

[29] R. Robbes. "Services". page on Squeak Swiki. [Online]. Available: http://minnow.cc.gatech.edu/squeak/3727