Type Harvesting

A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages

Michael Haupt Michael Perscheid Robert Hirschfeld
Software Architecture Group, Hasso-Plattner-Institut, University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

ABSTRACT

Dynamically typed programming languages are powerful tools for rapid software development. However, there are scenarios that would benefit from actual type information being available—e. g., code generation and optimisation as well as program comprehension. Since code written in such languages usually makes little or no explicit assumptions about types, type inference is not particularly well suited to obtain the desired information. This paper introduces *type harvesting*, a practical approach to obtaining type information. It is based on stepwise code execution of the code in question, closely observing the types of entities in question. Type harvesting allows for exploiting unit tests to automatically obtain type information for a code base. The approach has been implemented in Squeak/Smalltalk. Its evaluation, using several complex applications, shows that type harvesting yields excellent results with high precision.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development and maintenance*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Languages

Keywords

Dynamically typed programming languages, type inference, dynamic analysis, unit tests, type harvesting

1. INTRODUCTION

Dynamically typed programming languages are popular and widely used. In many application areas, they exhibit their benefits, allowing for quick turnarounds in development and high developer productivity. Still, it needs to be noted that dynamic languages have some shortcomings, especially when it comes to systems programming. In the respective areas, the development cycles in which dynamic languages are used impose a requirement that the languages cannot easily meet: static type information is required. For instance, the PyPy [17] virtual machine code generation tool chain requires complete static type information to be able to generate C code from the original Python code.

Intuitively, it is clear that complete and correct type inference is not easily achievable in dynamic languages: parts of language semantics must be provided as part of the inference engine, which leads to duplicated effort. Thus, when static type information is required, one needs to find other solutions. There exist inference engines [4, 2, 1, 20, 16, 6] that are however necessarily incomplete. There also exist lightweight approaches like *pluggable types* [3, 8], where developers can annotate methods with type information. Such annotations, however, are somewhat alien to dynamically typed languages and therefore do not invite acceptance.

We propose to instead extract type information from existing code by *harvesting* it during execution, when types naturally occur. Observing an application while running and introspecting its execution is typically not a large problem in dynamic languages, as they usually come with rich meta-programming facilities. These could be used to observe actual application code—but since the meta-programming efforts connected with harvesting are likely to impose a certain execution overhead, this may be unacceptable. Perhaps more appropriately, existing tests—in the form of unit or larger-scale integration tests—can be executed, and type information could be extracted from such runs. For this to work, though, good test coverage is a requirement.

This paper makes the following contributions. First, we propose the technique we call *type harvesting* as a pragmatic approach to obtaining type information without the burden of full-fledged and necessarily incomplete inference engines. A *type harvester* is an entity that gathers type information by observing application code while it is running. Rather than protocols or object structures, the harvested types are objects' *classes*, as these are of most interest in the application domains that we address (cf. Sec. 2).

Second, we present an implementation of type harvesting in Squeak[1] [12], an implementation of Smalltalk [7]. The harvester collects type information for method parameters, instance members, and all steps of the Squeak interpreter's

[1] `www.squeak.org`

```
greet: aString
  | s |
  s := WriteStream on: ''.
  s nextPutAll: 'hello, '; nextPutAll: aString.
  ^ s contents
```

Listing 1: Running example code.

bytecode execution. Users can control harvesting scope and granularity, collecting type information for any subset of the above points, and restricting harvesting to specific classes or packages. For each of these points of interest, the harvester can provide least specific types—with which we denote the least upper bound of all types occurring at a given point—as well as the entire bandwidth of types that occur there.

Third, the evaluation of type harvesting and the Squeak implementation discusses the concept in the context of several third-party applications. Type harvesting delivers excellent results in the evaluation. The discussion furthermore contributes implementation guidelines for porting type harvesting to other languages than Smalltalk.

In the remainder of this paper, we give a more detailed overview of uses of type information in dynamic programming languages in Sec. 2. Subsequently, in Secs. 3 and 4, we describe, evaluate and discuss type harvesting and its implementation in Squeak. In Sec. 5, we discuss related work on typing support in dynamic programming languages. Finally, Sec. 6 summarises the paper.

2. USES OF TYPE INFORMATION IN DYNAMIC PROGRAMMING LANGUAGES

Although dynamically typed programming languages offer many benefits, lacking type information can be disadvantageous. On the one hand, translators from dynamically typed high-level code to code in a statically typed but better optimisable language require type information to generate correct code. On the other hand, program comprehension is impeded and IDE tools are hindered in analysing the static properties of source code. Especially, programming language concepts that imply late binding are more difficult to understand and to follow when type information is missing.

2.1 Running Example

Throughout this paper, we will use a running example implemented in Squeak. Lst. 1 shows the source code of a method named *greet:*; Lst. 2 shows its bytecode instructions. The method is defined in the class `Greeter`. In Squeak, classes are organised in so-called *categories*, which group classes but do not provide namespaces. The category containing the `Greeter` class is named `Greetings`.

The method accepts one argument named `aString`. Since Smalltalk is dynamically typed, it is a common idiom to give method arguments names that hint at the expected types. In this case, the name denotes that the method expects an instance of the `String` class (or any of its subclasses).

The method has a temporary (local) variable named `s`. A `WriteStream` object is created and assigned to `s`. Next, the text "hello, " and the string passed to the method in `aString` are written to the stream. Finally, the method returns the stream's contents.

Further explanations on the method, its bytecode instructions and the problems that arise due to the lack of availability of type information will be given below.

```
37 <41> pushLit: WriteStream
38 <22> pushConstant: ''
39 <E0> send: on:
40 <69> popIntoTemp: 1
41 <11> pushTemp: 1
42 <88> dup
43 <24> pushConstant: 'hello, '
44 <E3> send: nextPutAll:
45 <87> pop
46 <10> pushTemp: 0
47 <E3> send: nextPutAll:
48 <87> pop
49 <11> pushTemp: 1
50 <D5> send: contents
51 <7C> returnTop
```

Listing 2: Running example bytecode instructions.

2.2 Code Generation

Several projects have chosen dynamically typed high-level programming languages to implement applications, which are then translated to lower-level languages before eventually being compiled to machine code. The rationale is that stand-alone static compilers, e. g., for C++, apply sophisticated optimisation mechanisms that yield excellent performance without imposing the overhead of a run-time environment.

The PyPy [17] project, for instance, provides an implementation of the Python virtual machine in Python. It employs a tool chain that generates C code from the high-level Python code. To facilitate this, full type information is required, which the PyPy tool chain obtains by applying type inference. Developers have to pay a price, though: instead of full Python, a dialect called RPython is used that somewhat limits the dynamic capabilities of Python.

Other projects have pursued similar goals without necessarily aiming for systems software [13, 15, 2], mostly applying source-to-source translation and type inference. Source-to-source translation requires building a compiler or at least exploiting existing parsing infrastructure for the dynamically typed source language.

We argue that using the result of the source language's compiler—e. g., bytecode instructions—eases target code generation as it avoids parsing. Also, it is not necessary to identify sub-expressions, as all bytecode instructions stand on their own. However, type information is still required.

For instance, consider the bytecode instruction at index 51 in Lst. 2; it returns the current top-of-stack value, which was returned from sending the `contents` message to the `WriteStream` created earlier. The type of this value is the return type of the method, but which one is it? Likewise, it is not necessarily obvious what type can be expected as the result of sending `on:` to the `WriteStream` class: will it return an instance of a specialised subclass, or a `WriteStream`?

While the developer can answer such questions by exploring the environment, a code generator needs to have access to reliable information to generate correct code. Typical idioms used in dynamically typed languages, such as naming arguments after their expected types (see above), do not help here as they cannot be enforced. Also, instance members' names typically do not adhere to such idioms. Thus, type information is required for all entities including the top-of-stack information at each bytecode instruction.

2.3 Program Comprehension

Program comprehension in dynamically typed languages can be improved with type information as it helps in navigating source code and using APIs correctly. Developers maintain a mental model of program behaviour [14] by navigating the static call graph for a specific method of interest and following several paths comprised of sender and implementor relationships. Such call graphs have various branches leading to many developers' decisions about the proper application of specific methods. Unfortunately, the set of possible paths is much larger in dynamically than in statically typed languages. For instance, identical method signatures in different classes yield ambiguous results—late binding hinders determination of actual methods since receiver object types are unknown until run-time. Additional type information reduces the set of call graph branches to the possibilities actually assigned in a specific context [10]. Thus, the developer's static source code navigation can be improved by selecting only those sender and implementor methods that relate to a specific type.

If type information is made available, it supports API usage as it makes interface descriptions more meaningful. Consequently, clients know what parameters to provide and what results to expect. Without type information, developers have to comply with conventions such as the one described above for the `aString` parameter, which is more error-prone as types cannot be checked before run-time. Available type information, as part of an interface contract, improves API robustness and correctness.

Based on additional type information, IDE tools can be improved by reducing result sets, checking for API conformance, and deducing concrete run-time behavior more precisely. This can help limiting the scopes of search, navigation and auto-completion tools to actually used types instead of all possible matching message signatures. Static analysis tools can check API usage to indicate related problems to give developers instant feedback about accidental mistakes while writing code. Finally, dynamic analysis tools can verify deduced type information to identify failure causes [24].

3. TYPE HARVESTING

In this section, we first give a high-level overview of the concepts of type harvesting, and of the building blocks required to put the approach to work. After that, we describe the implementation of type harvesting in Squeak.

3.1 Core Ideas

Type harvesting for dynamic programming languages gathers detailed type information from *live* systems, i. e., from running code. As pointed out in Sec. 2, the level of detail required from type information may be very high. Thus, type harvesting must be able to obtain type information practically at all points during execution—before or after the execution of each particular statement or even bytecode instruction. Likewise, it must be possible to confine harvesting activities to actually interesting parts of a software system.

We propose to fulfil these requirements by adopting an approach that applies *stepwise execution* and harvests type information at each step for the relevant system components.

```
(THHarvester forCategories: #( #'Greetings' ))
  eagerLeastSpecific;
  harvest: [ Greeter new greet: 'world' ];
  bucket
```

Listing 3: Sample type harvester invocation.

We call the set of all such components an *acre*. In Fig. 1, the principles of type harvesting are illustrated.

Given an acre, the harvester can be instructed to gather all type information found thereon. It initiates stepwise execution of the code under observation (index 1 in the figure). The stepwise execution logic is instructed to branch to the harvesting logic whenever it is about to execute a single instruction found on the acre. That way, the harvester is able to collect all relevant type information at each particular point in execution and store it in a data container called *bucket*. The bucket holds mappings from bytecode instructions, method arguments, local variables, return values, and instance members to their concrete types.

Once the harvesting logic has stored type information at a particular step, the current instruction is executed (index 2 in Fig. 1). In case this leads to the next instruction being found outside the acre—e. g., if the current instruction sends a message to an instance of a class outside the acre—, the next instruction is simply executed (index 3). As soon as the next on-acre instruction is found, harvesting continues.

The illustration in Fig. 1 shows a solution where particular instructions are executed *after* harvesting. This does not have to be the case; it is equally possible to harvest after execution. It is however important to take into account whether type crop represents results of the preceding or current instruction when harvesting results are used further.

3.2 Type Harvesting in Squeak

We have implemented *type harvesting* in Squeak. In the following, we will first describe the type harvesting API before presenting the implementation.

3.2.1 Type Harvesting Interface

Our type harvesting implementation provides an easy-to-use API that allows for fine-grained crop selection. The `THHarvester` class provides the entry point to harvesting functionality in the form of an embedded DSL. A typical invocation is shown in Lst. 3; it harvests type information from the example used in Sec. 2.1.

In the first line of the listing, a harvester is created; it is configured for harvesting *only* methods contained in the `Greetings` category. Effectively, this means that only the information for the `greet` method will be harvested. The next line instructs the harvester to be *eager*, i. e., to collect *all* available type information. The various options for this—and the messages that can be sent to `THHarvester` instances to control this—are listed in Tab. 1.

The `harvest:` message starts harvesting. The block that is passed with this message contains the code serving as the entry point to harvesting. All crop is gathered in a *bucket*, which is obtained in the last line of Lst. 3.

Depending on whether the harvester collects *all* or just *least specific* types, the `bucket` message will answer a `THPolyBucket` or `THMonoBucket` instance. The different buckets can be queried for all the crop they contain. The `TH-`

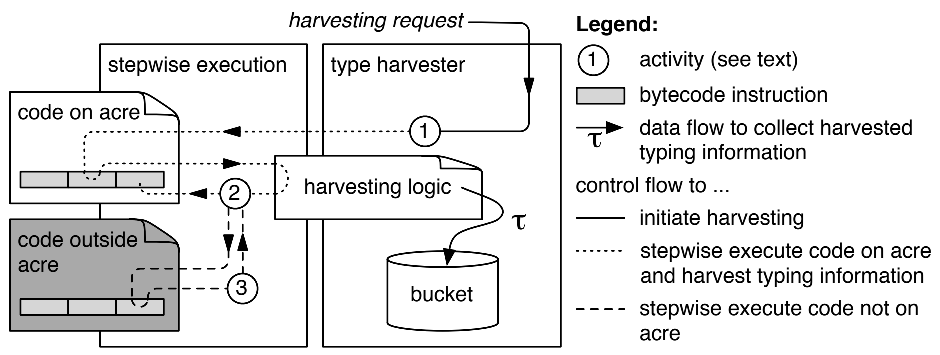

Figure 1: Illustration of the type harvesting approach.

option message	harvest ...
`arguments`	method arguments types
`topOfStack`	top-of-stack types
`members`	instance member types
`returns`	method return types
`stackp`	stack depths
`temps`	temporary ("local") variable types
`allTypes`	collect *all* types occurring at the above points, instead of just the least specific type
`eager`	all of the above
`eagerLeastSpecific`	all of the above, least specific type only

Table 1: Options for `THHarvester` instances.

`PolyBucket` will always answer sets of classes occurring at the respective points; the `THMonoBucket`, single classes.

Sending the query `typeAt: 51 in: Greeter >> #greet:` to the bucket resulting from the example in Lst. 3 will yield the least specific type (class) ever encountered prior to the execution of the return bytecode instruction in the `greet` method (`ByteString`). The same result can be obtained by sending the query `returnTypeFor: Greeter >> #greet:`.

3.2.2 Implementation Details

The Squeak standard image comes with a *bytecode simulator* that can be used to execute code in simulated execution; i. e., there is an entire Smalltalk bytecode interpreter at image level. The simulator also allows to intercept execution at each step and apply some additional functionality.

For instance, the code shown in Lst. 4 interprets the code already used in the example in Sec. 2.1 step by step. Prior to the execution of each single bytecode instruction, it will print the current method and program counter. Note that this holds for *all* methods called during the execution of the sample code: the complete dynamic extent of the first argument block is traced.

The `ctx` argument passed to the second block (the "context block") in the example is an instance of `MethodContext` representing the current execution context at the time the block is invoked. This object provides access to all details of the running code, including the call stack, method being executed, program counter, local variables, etc.

```
thisContext
  runSimulated: [ Greeter new greet: 'world' ]
  contextAtEachStep: [ :ctx |
    Transcript
      show: ctx method printString;
      show: ctx pc printString;
      cr ]
```

Listing 4: Simulated bytecode execution in Squeak.

In our implementation, the context block first checks if the currently executed method is on the acre; i. e., whether it is contained in one of the categories the harvester was created for. If so, the harvester executes the actual harvesting logic, filling the bucket with the desired crop.

The on-acre check needs to be performed at *all* bytecode instructions, which obviously imposes a performance impact on execution. To reduce the overhead, the harvester applies a caching scheme that avoids expensive on-acre checks as long as the method being executed stays the same. We discuss the performance impact of type harvesting in Sec. 4.

4. EVALUATION

Type harvesting allows for an extensive and fine-grained collection of type information. We evaluate our approach by comparing harvested with used types and by measuring the required execution time for collecting dynamic type information. Detailed results are available online.[2]

4.1 Experimental Setup

We selected four different Squeak projects to analyse type harvesting with respect to effectiveness and efficiency; i. e., how good the quality of harvested types is, and how long harvesting takes. One of the four projects, AweSOM [9], is a research project, namely a virtual machine for a Smalltalk dialect (SOM Smalltalk) written in Squeak. There exist implementations for SOM Smalltalk in Java, C, and C++. The SOM Smalltalk standard library and an acceptance test suite are shared between all four SOM implementations.

The remaining three projects are daily used software development tools and parts of the Squeak standard distribution, namely the Smalltalk system browser, an XML parser, and the SUnit unit testing framework.

The project properties are summarised in Tab. 2. The

[2]To reviewers: the results are available via the PC chairs.

	AweSOM acceptance tests	AweSOM unit tests	System Browser	XML Parser	SUnit
Classes	68		51	25	16
Methods	742		1,298	367	312
Tests	14	124	62	8	41
Method coverage	77.9 %	77.6 %	33.0 %	32.0 %	64.0 %

Table 2: Summary of project properties.

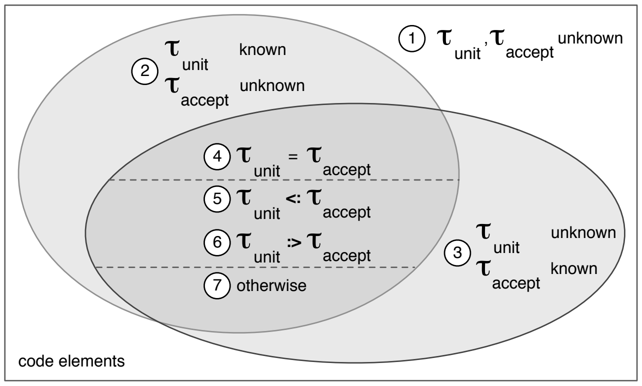

Figure 2: Type sets for effectiveness evaluation.

complexity of all projects is typical of small- and mid-sized Smalltalk systems and they include a total of 249 tests including system, acceptance, and unit tests.

We measured harvesting effectiveness (see Sec. 4.2) with the help of AweSOM and its two independent test suites and the efficiency (see Sec. 4.3) by profiling all five test suites in normal, simulated and type harvesting mode. All experiments were run on a MacBook with a 2.4 GHz Intel Core 2 Duo and 4 GB RAM running Mac OS X 10.6.4, using Squeak version 4.1 on a 4.2.1b1 virtual machine.

4.2 Effectiveness

We evaluate the quality of harvested information by comparing the type sets of two independent test suites. AweSOM possesses both unit tests, which verify the virtual machine functionality from the implementation perspective, and acceptance tests, which reflect typical uses of the system. The acceptance suite is written in SOM Smalltalk.

During the execution of both test suites, we harvest unit test types (τ_{unit}) as the measured set and acceptance test types (τ_{accept}) as the reference set. Since the acceptance tests imply actual runs of the complete VM, they represent "real-world" scenarios from the perspective of code coverage. Comparing both types at each code element (i. e., member variable, local variable, method return, top-of-stack at instruction) allows for a qualitative assessment of harvesting results. After type harvesting, each code element has up to two possible types leading to seven disjoint type relationship sets as shown in Fig 2:

1. The code element was not covered by any test. No type information is available.
2. Only the unit test type (τ_{unit}) is available. No acceptance test has covered this element.
3. Only the acceptance test type (τ_{accept}) is available. No unit test has covered this element.
4. Both types are available and identical (perfect match).
5. Both types are available but τ_{unit} inherits from τ_{accept} (imperfect match).
6. Both types are available but τ_{accept} inherits from τ_{unit} (imperfect match).
7. Both types are not compatible to each other.

For good harvesting effectiveness, as much type information as possible must be collected (*coverage*) including most of the reference set (*recall*), and harvested types should be identical or at least, via inheritance, related to each other (*precision*). We define the three metrics (coverage, recall and precision referring to information retrieval [23]) based on the type relationships of code elements as follows (numbers denote the *sets* from above and illustrated in Fig. 2):

$$\text{coverage} = \frac{|\tau_{unit} \text{ available}|}{|\text{code elements}|} = \frac{\left|\bigcup_{k=2,4,5,6,7} k\right|}{\left|\bigcup_{k=1}^{7} k\right|}$$

$$\text{recall} = \frac{|\tau_{unit} \text{ and } \tau_{accept} \text{ available}|}{|\tau_{accept} \text{ available}|} = \frac{\left|\bigcup_{k=4}^{7} k\right|}{\left|\bigcup_{k=3}^{7} k\right|}$$

$$\text{precision} = \frac{|\tau_{unit} = \tau_{accept}|}{|\tau_{unit} \text{ and } \tau_{accept} \text{ available}|} = \frac{|4|}{\left|\bigcup_{k=4}^{7} k\right|}$$

Coverage is the extent to which the system includes harvested type information. To maximise coverage, the first and third set must be minimised as they do not include type information from unit tests. *Recall* is the proportion of harvested (measured set) and used-in-reality type information (reference set). To ensure a high value the third set should be as small as possible. Last but not least, *precision* is the proportion of retrieved types that are actually identical[3]. This metric evaluates the quality of type matches. For that reason, the sets 5–7 should be small in comparison to set 4. Especially, the seventh set should be nearly empty as types contained therein are incompatible to each other.

The results of our experiments are summarised in Tab. 3. It presents the absolute sizes of all subsets and the computed values for coverage, recall and precision.

Type Sets.

The sets (1)–(3) show that some portions of the system are only partially covered. This was to be expected given a method coverage of about 78 % (cf. Tab. 2). Moreover, it can

[3]At this point only perfect matches influence precision; a discussion about imperfect ones can be found in Sec. 4.4

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	Coverage	Recall	Precision
Member variables	22	0	0	62	0	2	0	74.42 %	100.00 %	96.88 %
Local variables	97	38	29	378	10	26	0	78.20 %	93.45 %	91.30 %
Method return	124	19	29	288	4	10	0	67.72 %	91.24 %	95.36 %
Top-of-stack at instruction	1,320	400	340	3,950	57	104	1	73.10 %	92.36 %	96.06 %
All type information	1,563	457	398	4,678	71	142	1	73.17 %	92.48 %	95.63 %

Table 3: Type harvesting effectiveness. Numbers (1)–(7) represent type subsets (see Fig. 2).

be seen that the parts covered by unit or acceptance tests *only* (sets (2) and (3)) have about the same sizes, which are small compared to the total. Set (4), representing perfect matches, is the most important one, and also the largest.

Sets (5) and (6) represent imperfect matches. Even though their overall number is low, a size comparison shows that unit tests (set (6)) deliver less specific classes twice as often as acceptance tests. This indicates that unit tests tend to be more fine-grained in that they test more alternatives than might be needed in "real-world" acceptance test scenarios. Looking at the detailed results, it can be noted that the same combinations of classes repeatedly occur: e. g., `OrderedCollection` and `SequencableCollection` (a test checks more than just the actually used collection), or `False` and `Boolean` (a test does not cover one conditional branch). These cases are good suggestions for providing more and better tests.

The seventh set is ideally empty. In our case, one conflict was found, where the unit test harvested a `ByteSymbol`, and the acceptance test, a `SOMClass`. The two are unrelated save their inheritance relationship to `Object`. Closer investigation of the code revealed that the unit test used a mock object, yielding a type that would never be used in practice.

Coverage Evaluation.

The coverage results in Tab. 3 are close to the overall method coverage of roughly 78 % obtained during normal test execution (cf. Tab. 2). This is acceptable, as type information could be harvested for large portions of the system.

The observed difference of up to 10 % in the results are due to differences in computing coverage. In particular, coverage of member variables is obtained differently than that of methods. Method return values may be uncovered because non-local returns occur. Finally, so-called *quick methods* (usually, simple getters) are used to optimise performance: such methods are flattened into simple bytecode instructions and thus do not appear to the harvester. AweSOM has over 100 of such quick methods. Their return types can easily be derived by observing the types of the members they return.

Evaluation of Recall and Precision.

Recall and precision are crucial for evaluating harvesting quality, as they represent types used in reality, and the accuracy of identified matches. The results obtained for these metrics are very good, at over 90 %. For *recall*, this means that most of the types used in reality are actually found by unit tests. For *precision*, we can conclude that when a type is harvested in a unit test, it will very likely occur in real code. Especially this metric should be close to 100 %, as such a high value indicates to developers that harvested type information is in accordance with reality. Although both metrics do not completely reach 100 %, they are very satisfactory. They can be improved by providing more tests.

Summary.

All three metrics—coverage, recall, and precision—should yield high values to ensure good quality of results. In particular, recall and precision should be close to 100 % as they best reflect in how far harvested type information is in accordance with reality. On average, for all harvested types, we obtain a coverage of 73.17 % (i.e., all possible types), a recall of 92.48 % (i.e., overlapping of harvested types), and a precision of 95.63 % (i.e., identically harvested types), which underlines applicability and feasibility of our approach.

4.3 Efficiency

We evaluated the efficiency of our implementation by running four Squeak projects and measuring their performance. Tab. 4 summarises the required absolute run-times and related execution overheads. Normal execution by the VM is in the range of some seconds for all four projects. Running the code in simulation mode slows it down by 140–270 times, arriving at several minutes' execution time. The additional overhead induced by harvesting is comparatively low, with some 40 % on average (the XML parser project exhibits, due to its very small execution time, a larger impact of harvesting "noise" than the other projects).

As type harvesting requires fine-grained stepwise execution, the resulting overall execution time in harvesting mode is on the scale of minutes. One typical application scenario of type harvesting is code generation, where performance is not crucial. Another scenario, however, is program comprehension, where quick feedback might be important. Therefore, we consider the investigation of quicker implementations—e. g., selectively simulating bytecode execution for points of interest only—an important area of future work.

4.4 Discussion

In this section, we comment on various aspects of the observations made above, and also discuss the approach.

Threats to Validity.

Good test coverage is a core requirement for our approach. We argue that the observed 78 % are sufficient, as the acceptance test suite used in the evaluation has been used with several SOM implementations so far and challenges the VM implementation in several regards. The very high recall value also shows that the unit tests largely cover the acceptance tests; their quality is thus rather high. The remaining uncovered 22 % of AweSOM largely consist of code not used in reality; e, g., legacy implementation artifacts or helper methods for debugging purposes.

We have evaluated effectiveness in the context of only one project. However, this particular project is very well suited for such a task for three reasons. First, there are two independent test suites provided by different authors. The acceptance test suite consists of a collection of SOM Smalltalk

	AweSOM		System	XML	SUnit
	acceptance tests	unit tests	Browser	Parser	
normal execution	3.5 s	2.5 s	0.8 s	0.005 s	3.3 s
simulated execution	500.3 s	438.3 s	114.0 s	1.6 s	657.8 s
times slower than normal	143x	175x	143x	272x	199x
type harvesting	704.0 s	603.0 s	128.0 s	5.4 s	727.8 s
times slower than normal	201x	241x	160x	899x	220x
times slower than simulated	1.41x	1.38x	1.12x	3.29x	1.10x

Table 4: Summary of run-time measurements and overhead.

applications that are executed by the VM: they are *real-world* code whose execution involves the entire VM. Next, the domain of a VM implementation brings about several different types. Finally, there exist other implementations of the SOM VM, even in statically typed languages (Java and C++). Still, regarding other application domains for further studies is an important aspect of future work.

Efficiency was assessed using four different projects; the weak performance results are mostly due to simulated stepwise execution. We do not expect the observed performance characteristics to differ when other projects are used.

The recall and precision metrics could easily be manipulated to reach 100 %. For recall, all code elements could be assigned the type `Object`; for precision, similar "hacks" are conceivable. However, such "optimisations" would make the respective other metric drop to almost zero. We thus regarded only perfect matches as relevant for precision and dealt with imperfect matches differently. For the sake of completeness, we would like to point out that `Object` was harvested as least specific type in only 2.3 % of all cases. Upon closer observation, these cases are unproblematic, e. g., when elements in collections may have arbitrary type.

Improving Precision.

Although precision is rather high already, we have only regarded *perfect matches* (set (4)) so far to obtain this value. However, imperfect matches with inheritance relationships (sets (5) and (6)) might be considered worthwhile as well depending on application scenarios.

First, if a less specific type is harvested from a unit test (set (6) is relevant), the result can still be used for program comprehension. It indicates that all subclasses used in reality could occur at the observed code entities; e. g., if `SequencableCollection` is harvested, and `OrderedCollection` or `SortedCollection` are used in reality.

Second, if acceptance tests deliver less specific types than unit tests (set (5) is relevant), it can be guaranteed that the protocol harvested from acceptance tests is always valid for subclasses. This is important for code generation, since the least specific type can safely be used for interfaces.

Taking these usage scenarios into account, we can obtain higher precisions of 98.5 % (with set (6), for program comprehension), or 97.0 % (with set (5), for code generation).

Increasing Performance.

Simulated bytecode execution allows for collecting fine-grained type information, but it induces a significant slowdown. The bottleneck is however simulation itself, not harvesting. The latter has a modest overhead. Future work will investigate alternative approaches, e. g., inserting hooks into the VM's bytecode interpreter itself. Also, the observed performance issues will likely be mitigated as overall run-time environment performance increases. For Squeak, a just-in-time compiler named *Cog*[4] is currently being developed that promises significant overhead reductions. We are collaborating with the author to improve its robustness, which is currently not sufficient to serve as the basis for our work.

It needs to be noted that it is not necessary to always run a full test suite to obtain type information. In fact, there exist approaches that adopt incremental execution of relevant tests as those parts of an application covered by them are changed [19, 21]. Integrating type harvesting with such approaches is another subject of future work.

Scalability to Other Languages.

A type harvesting implementation requires a fine-grained stepwise execution mechanism such as Squeak's simulation engine. Some languages provide support for similar techniques; e. g., Python's interpreter hook, or Java's debugging interface (which, given that Java is statically typed, could be used for Groovy). In other languages that do not support such an approach, extensions of the virtual machine or meta-programming might be applied.

5. RELATED WORK

We restrict the discussion of related work to approaches that allow for obtaining type information in dynamically typed languages. Consequently, there are two main categories to consider: type inference and run-time type collection. Apart from these two, pluggable types [3, 8] are worth mentioning, but since they require developers to provide type information, we do not discuss them here.

Type inference for dynamically typed programming languages has been researched early on [22]. More recent approaches [4, 2, 1, 20, 16, 6] mostly focus on obtaining type information for interfaces (i. e., method arguments and return types) and member as well as local variables. They do not deliver the fine-grained results type harvesting offers. Moreover, their precision is sometimes lower [20, 16].

Run-time type collection is scarcely used in a way that allows programmers to exploit its results. Type feedback [11] is a VM technique that drives just-in-time compiler optimisation decisions based on dynamically collected type information. This information is confined to the VM. Type inference for the Ruby programming language [6] has been augmented by profile-guided typing [5]. This approach requires the code under observation to be instrumented—unlike type harvesting, which leaves code on the acre untouched. As pointed out for type inference approaches above, type information is also not as fine-grained as with harvesting.

[4] `www.mirandabanda.org/cogblog`

The Hermion [18] IDE extension for Squeak is designed for providing developers with additional run-time information during static source code navigation. Hermion, among other things, improves IDE usability by exploiting dynamic type information, effectively resolving navigation issues related to late-binding. For instance, Hermion restricts senders and implementors lists in Smalltalk to relevant run-time types only. To that end, it permanently collects method signatures and receiver types during normal program use. Conversely, type harvesting can be executed on demand, and harvested information is more extensive.

6. SUMMARY

We have introduced *type harvesting*, a practical approach to obtaining type information in dynamically typed programming languages from stepwise execution of code. Type harvesting yields excellent results with high precision, which can be exploited in various tasks, such as code generation and program comprehension.

Ongoing and future work are concerned with performance improvements as well as putting type harvesting to use in the aforementioned domains. Moreover, we are investigating how harvesting results can be used for other purposes, such as detection of type violations and potentially dead code or automated suggestion of test cases.

Acknowledgements

We are grateful to Eliot Miranda for fixing a critical defect in the Squeak bytecode simulator. We thank Carl Friedrich Bolz and the reviewers for their comments on this paper.

7. REFERENCES

[1] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of self: analysis of objects with dynamic and multiple inheritance. *Softw. Pract. Exper.*, 25(9), 1995.

[2] O. Agesen and D. Ungar. Sifting out the gold: delivering compact applications from an exploratory object-oriented programming environment. In *Proc. OOPSLA'94*, pages 355–370. ACM, 1994.

[3] G. Bracha. Pluggable Type Systems. OOPSLA 2004 Workshop on the Revival of Dynamic Languages, `http://bracha.org/pluggableTypesPosition.pdf`.

[4] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proc. OOPSLA'93*, pages 215–230. ACM, 1993.

[5] M. Furr, J.-h. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proc. OOPSLA'09*, pages 283–300. ACM, 2009.

[6] M. Furr, J.-h. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *Proc. SAC'09*, pages 1859–1866. ACM, 2009.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[8] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, pluggable types for a dynamic language. *Computer Languages, Systems & Structures*, 35(1):48–62, 2009. ESUG 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007).

[9] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn. The SOM Family: Virtual Machines for Teaching and Research. In *Proc. ITiCSE'10*, pages 18–22. ACM, 2010.

[10] R. Holmes and D. Notkin. Enhancing Static Source Code Search With Dynamic Data. In *SUITE'10.* ACM, 2010.

[11] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *SIGPLAN Not.*, 29, 1994.

[12] 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 *Proc. OOPSLA'97*, pages 318–326. ACM Press, 1997.

[13] T. Kishi, R. Shinozaki, M. Hama, H. Sawada, and M. Kagawa. Application development using Smalltalk and C++. In *Proc. COMPSAC'91.* IEEE, 1991.

[14] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. ICSE'06*, pages 492–501, New York, NY, USA, 2006. ACM.

[15] C. Nash and W. Haebich. An accidental translator from smalltalk to ansi c. *SIGPLAN OOPS Mess.*, 2(3):12–23, 1991.

[16] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *Proc. DLS'09*, pages 69–78. ACM, 2009.

[17] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM Press.

[18] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting Runtime Information in the IDE. In *Proc. ICPC'08*, pages 63–72. IEEE Computer Society, 2008.

[19] D. Saff and M. Ernst. Continuous testing in Eclipse. *Electronic Notes in Theoretical Computer Science*, 107:103–117, 2004.

[20] S. A. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proc. ECOOP'04.* Springer, 2004.

[21] B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld. Continuous selective testing. In *Proc. XP'10.* Springer, 2010.

[22] N. Suzuki. Inferring types in smalltalk. In *Proc. POPL'81*, pages 187–199. ACM, 1981.

[23] C. van Rijsbergen. *Information Retrieval.* Butterworth, 1979.

[24] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, 2006.