Practical, pluggable types for a dynamic language ☆

Niklaus Haldiman, Marcus Denker*, Oscar Nierstrasz

Software Composition Group, IAM, Universität Bern, Switzerland

ARTICLE INFO

ABSTRACT

Most languages fall into one of two camps: either they adopt a unique, static type system, or they abandon static type-checks for run-time checks. Pluggable types blur this division by (i) making static type systems optional, and (ii) supporting a choice of type systems for reasoning about different kinds of static properties. Dynamic languages can then benefit from static-checking without sacrificing dynamic features or committing to a unique, static type system. But the overhead of adopting pluggable types can be very high, especially if all existing code must be decorated with type annotations before any type-checking can be performed. We propose a practical and pragmatic approach to introduce pluggable type systems to dynamic languages. First of all, only annotated code is type-checked. Second, limited type inference is performed on unannotated code to reduce the number of reported errors. Finally, external annotations can be used to type third-party code. We present TypePlug, a Smalltalk implementation of our framework, and report on experience applying the framework to three different pluggable type systems.

© 2008 Elsevier Ltd. All rights reserved.

1. Introduction

Any static type system attempts to reduce the number of errors that may occur at run-time by restricting the programs that one is allowed to write. Statically typed programming languages implicitly take the position that this is a good deal. In return for the minor inconvenience of not being able to write certain kinds of relatively uncommon programs, we obtain guarantees that no catastrophic run-time errors will occur.

Dynamically typed languages assume the opposing standpoint. Precisely those interesting programs—for example, those exploiting behavioural reflection—would be rejected. Furthermore, although there are many different approaches to static typing, a statically typed programming language must commit to one. Dynamic languages do not need to make this kind of premature commitment. Finally, static type systems can generate a false sense of security. Although certain kinds of catastrophic errors can be detected, others cannot.

Gilad Bracha has argued that it is possible to have one's cake and eat it too [1]. Instead of static type systems being mandatory, they should be *optional.* This means that a type system should neither require syntactic annotations in the program source, nor should it affect the run-time semantics of the language. An optional type system would report possible errors, but would not prevent the program from being run (and possibly failing at run-time). In addition, type systems should be *pluggable*, that is, one should be able to choose the kind of static checks one would like to perform. This is especially interesting given the relatively recent emergence of more exotic type systems for reasoning about confinement [2], aliasing [3], scopes [4] and so on.

A number of pluggable type systems have been proposed over the last years [5–7]. The key difficulty with pluggable types, however, is a practical one: *In the presence of a large, existing software base, how can one benefit from a pluggable type system without annotating the legacy code?*

We present a practical and pragmatic approach to pluggable types that significantly reduces the overhead of adopting a new type system. TypePlug is a framework supporting pluggable types implemented in Squeak Smalltalk. Type systems are defined by specializing the framework. Types are declared using an annotation framework. The contribution of TypePlug is to reduce the cost of adopting a type system by means of the following techniques:

(1) Only annotated code is type-checked.
(2) Type inference is applied to unannotated code to reduce propagation of errors.
(3) Explicit type casts shortcut type-checking.
(4) External annotations are supported to declare types for third-party code.

Section 2 provides a brief introduction to TypePlug using an example of a non-nil type system. In Section 3 the TypePlug framework is described in some detail. Section 4 discusses experience using TypePlug to define a class-based type system and a confinement type system. Section 5 discusses the results obtained in the context of related work. We conclude in Section 6 with some remarks on future work.

This article is an extension of our previous work [8]. The current paper adds an in-depth presentation of the confinement type system (Section 4.2).

2. TypePlug in a nutshell

TypePlug provides a framework to optionally annotate code with type declarations and to type-check annotated code. We will demonstrate how optional types and type checking can be used in a typical coding session. All code examples shown are using the Squeak dialect of Smalltalk [9] extended with a syntax for textual annotations.

To keep the presentation as simple as possible, we use the example of a non-nil type system. This type system has exactly one type, the nonNil type. If a variable has the nonNil type it cannot hold the value nil. Anything not typed nonNil is considered to potentially evaluate to nil (there is no explicit nil type).

To declare a variable to be of type nonNil we add the annotation **<:nonNil :>** to it. In the following class definition of a two-dimensional line we declare the instance variable endPoint to be nonNil:

```
Object subclass: #Line
 uses: TReflectiveMethods
 typedInstanceVariables: 'startPoint endPoint <:nonNi :>'
 typedClassVariables:''
```

Notice the trait[1] TReflectiveMethods that we use in this class. It imports the annotation framework that we need to annotate its methods. We assume that we have not added any other annotations yet to the methods of this Line class. If we now browse the source of one of the methods we can experience type checking in action. Here is a method of a line that moves it horizontally by a number of units:

```
moveHorizontally: anInteger
 startPoint : = self movePoint: startPoint horizontally: anInteger.
 endPoint : = self movePoint: endPoint horizontally: anInteger ← type
  'TopType' of expression is not compatible with type 'nonNil' of variable 'endPoint'
                                                                                    .
```

When the source of a method is displayed it is type-checked and type errors are shown inline. The type error found in the above method is highlighted. What it says is that the expression self movePoint: endPoint horizontally: anInteger was found to have the top type which means that its type is *unknown*. Since the instance variable endPoint requires the type nonNil, this results in a type error.

To fix this error we have to examine the method movePoint:horizontally: and declare its return type to be nonNil. This is what it looks like:

```
movePoint: aPoint <:nonNil:> horizontally: anInteger
    ↑ (aPoint addX: anInteger y: 0) <:nonNil:>
```

A return type annotation must be added to the return statement expression. Notice how parentheses are needed here to apply the annotation to the whole expression. Adding a return type to a method has two repercussions: in call sites of the method the message send expression will be typed accordingly and within the method its return statements will be type-checked. In this

---

[1] A trait is a set of methods that can be included in the definition of a class [10].

example it means that the expression aPoint addX: anInteger y: 0 must be of type nonNil (assume we annotated the addX:y: method with a nonNil return type already, so no type error occurs here).

To show an example of an argument type we have annotated the first argument of movePoint:horizontally: with a nonNil type as well. Adding an argument type declares the argument to be of that type within the method, and it also poses a requirement to arguments passed when the method is used. Because of this requirement we now get a different type error when we look at moveHorizontally: again:

```
moveHorizontally: anInteger
  startPoint : = self movePoint: startPoint horizontally: anInteger ← in message
  'movePoint:horizontally:' of class 'Line' argument 'TopType' is not
  compatible with expected type 'nonNil'.
  endPoint : = self movePoint: endPoint horizontally: anInteger.
```

The startPoint instance variable does not have an annotation so its type is the default, the top type, which is not what is required for the first argument to movePoint:horizontally:. The obvious fix to this type error is to declare startPoint to be nonNil as well. A different strategy that TypePlug offers is to *cast* an expression to an arbitrary type, in this example by adding the annotation **<:castNonNil :>**.

```
moveHorizontally: anInteger
  startPoint : = self movePoint: startPoint <: castNonNil:> horizontally: anInteger.
  endPoint : = self movePoint: endPoint horizontally: anInteger.
```

With a cast a programmer asserts to the type checker that the type of a given expression is known. Casts are useful to resolve the type of errors where more type annotations would be overkill or impossible, for example because the cast expression uses untyped third-party code.

Let us annotate one more method of Line with a return type:

```
hasStartPoint
  ↑ (startPoint notNil) <:nonNil:>
```

We are still operating under the assumption that no type annotations have been made other than those previously discussed. We surely have not annotated any notNil method with types, so how come we do not get a type error here because the expression startPoint notNil cannot be proven to be nonNil? The reason is that the type checker tries to infer return types of methods if they are not explicitly annotated. The notNil method has two implementations, one in the base class Object—returning false—and one in UndefinedObject—returning true. So obviously invocations of notNil will always return a nonNil boolean value, which is what the type checker figured out in this case, sparing us from explicitly annotating the notNil method with a return type.

While this preceding coding session serves as a lightweight introduction to TypePlug , it also illustrates an important issue when programming with optional types. Once one adds just one type to previously untyped code the need usually arises for more type annotations to type-check other parts of the code. Most likely, these additional types prompt for even more type annotations. This means that quite a bit of work is usually required from a programmer if he introduces types in a section of his code. A framework for optional typing should be aware of this problem and minimize the work required by the programmer. As demonstrated in the coding session, two of our strategies in this area are type inference and casts, reducing the need for explicit type annotations.

3. The TypePlug framework

A type system in TypePlug is specified by first defining the types of the system and then the properties of the types. This mainly consists of defining mappings from Smalltalk elements to types as well as operations on types. Since a key goal of TypePlug is to enable the creation of new type systems without much effort, the number of properties that have to be defined is kept to a minimum while retaining considerable flexibility.

Concretely, a new type system is created by subclassing the TPTypeSystem class and overriding some of its methods. As a quick reference and overview, Table 1 lists all of the methods of TPTypeSystem that can be overridden when implementing a new type system. In the following discussion we will touch on the details of the most important of these methods.

3.1. Defining a type system

3.1.1. Persephone

Persephone [11,12] brings sub-method reflection to Squeak. The standard model of Squeak and generally any Smalltalk system does not provide a model for sub-method structure. Methods are just represented as text and bytecode.

Persephone enhances Squeak with a model for methods based on an abstract syntax tree (AST). Any entity of the AST can be annotated with metadata. Annotations can be visible in the source code, they can be added in various places in a method,

Table 1
Methods of TPTypeSystem to override in subclasses

systemKey	Unique key for this system. *Defined as class method*
annotationValueToType:inContext	Converts annotations to types. *Abstract, must be overridden*
is:subtypeOf	Defines the subtyping relation. *Abstract, must be overridden*
unifyType:with	Defines the type unification operation. *Abstract, must be overridden*
typeForArray	Maps arrays to types
typeForBlock	Maps blocks to types
typeForGlobal:value	Maps global variables to types
typeForLiteral	Maps literals to types
typeForPrimitive	Maps primitives to types
typeForPrimitiveNamed:module	Maps named primitives to types
typeForSelfInClass	Maps self pseudo-variables to types
typeForSuperInClass	Maps super pseudo-variables to types
methodsForMessage	Customizes the set of methods to be considered when type checking message sends
transformMethodType	Transforms methods type before they are used in the type checker
transformType	Transforms types before they are used in the type checker
assignmentTo	Customizes type checking for assignments
displayClass	Creates a custom string version of classes
displayMethod:in	Creates a custom string version of methods

namely to instance and class variables, method arguments, return statements, local variables and block variables. For example, the expression anExpression **<:aSelector: anArgument:>** attaches an annotation with one argument to the expression anExpression.

The argument of an annotation can be any Smalltalk expression. In the simplest case, when the argument is just a literal object, the value of the annotation is set to this literal object. When the argument is an expression, the value of the annotation is the AST of this expression. We can specify when this AST is evaluated, either at compile time or later at run-time. In addition we provide a reflective interface to query and set annotations at runtime.

3.1.2. Annotations, types and their representation

TypePlug gathers type information from annotations made in Squeak source using the Persephone framework. Such an annotation places the requirement that the annotated subexpression should have the given type.

In addition to type annotations TypePlug supports *cast annotations*. Casts are unchecked—they always succeed, regardless of whether they can statically be proven type safe. Cast annotations are formed by preceding a type annotation with cast and capitalizing its first character, e.g. **<:castNonNil :>** is a cast to type nonNil.

Since TypePlug supports an arbitrary number of coexisting pluggable type systems, any expression can have several annotations, one per type system. In the source code, annotations are simply added one after the other. For this to work we need a way to distinguish annotations for different type systems. Every type system defines a unique *system key* that is used to identify its annotations. The system key is a symbol returned from the class method systemKey.

For the non-nil type system, the key is nonNil, thus an annotation for the unique type in the system is **<:nonNil :>**. This is a special case—usually a type system will have more than one single type, and keys will usually have a colon at the end to signal that. For example, the confinement type system discussed in Section 4.2 has the key confine:. In an annotation for that system, the actual type appears as the *annotation value* after the key, e.g. **<:confine: toClass :>**.

The type system must define its valid annotation values and thus its valid types. It does so by defining a method annotationValueToType:inContext: which returns instances of its types (types in TypePlug are instances of subclasses of the class TPType). The annotationValueToType:inContext: method takes as the first argument an AST of such an expression. This is a great help for type systems which might have complex annotation values, so they do not need to do their own parsing. The second argument is an instance of TPContext which describes the context of the annotation, i.e. in which method the annotation at hand is located. In a type system, the interpretation of an annotation might depend on where it is found.

3.1.3. The top type

One type is predefined and shared by all type systems: the top type. It is assumed to be a supertype of every other type in a type system. Within the TypePlug framework, the top type can appear in many places where a type is expected, and if it does it means one of two things:

(1) This can be any type.
(2) Nothing is known about this type.

An example of the first meaning is the default argument type for methods: by default, if a method argument is not explicitly annotated with a type, it is assumed to have the top type. The second meaning can be observed in the typing methods discussed in the next section: all of these by default return the top type to state that, e.g. the type of literals (defined by typeForLiteral:) is unknown—unless, of course, they are defined differently by the type system.

Generally speaking, every typed thing has the top type if it is unannotated or has an unknown type. This may seem like an insignificant implementation detail but the top type is an important device to usefully type check only partially annotated source. With its property of being a supertype of every other type it guarantees that, e.g. unannotated source code type-checks safely.

3.1.4. Typing elements of Smalltalk syntax

Type systems can define the types of a range of basic constructs of Smalltalk syntax: types for literals, global variables, primitives, arrays, blocks and pseudo-variables such as self and super. This is achieved by implementing any of the typeFor* family of methods defined on type systems, as listed in Table 1. By default all of these methods return the top type.

In the non-nil type system, the implementation of the typing methods is very simple. Here are two of the more interesting ones:

```
typeForLiteral: aValue
 ↑ aValue ifNotNil: [self singleType] ifNil: [self topType]

typeForSelfInClass: aClass
 ↑ (UndefinedObject includesBehavior: aClass)
  ifTrue: [self topType]
  ifFalse: [self singleType]
```

Obviously, the type for a literal is always nonNil except when the value is nil. To assess the type of self we need to be slightly cautious since within nil's class, UndefinedObject, we cannot say that self is nonNil. But the same is true for all superclasses of UndefinedObject since their methods could be called from the nil instance (the includesBehavior: method returns true if the argument is the receiver or a superclass of the receiver).

3.1.5. Subtyping and unification

The two most important operations a type system must define in our model are a subtyping relation and a type unification operation. Each of these is heavily used at the core of the type-checking algorithm as described in Section 3.2. Responsibilities for subtyping and unification are assigned to the type system rather than to types, though the implementation of a particular type system may delegate these responsibilities to the types themselves.

The subtyping relation is defined by implementing the is:subtypeOf: method which takes two types as arguments. It should return true if the first type is a subtype of the second in the context of this type system, false otherwise. While type systems are free to define whatever subtyping relation they please, it should usually be reflexive and transitive to be of practical use. The second argument may be the top type, representing an unknown type. The top type is considered to be a supertype of every other type of any type system by definition, so is:subtypeOf: is never called with the first argument being the top type.

The unification operation creates a type that represents the union of two types. Again, type systems are completely free to define this in any way that is appropriate. Generally, the union of two types should be the most specific common supertype, but a type system can also work with an explicit union type that it defines. In code, the unification operation is defined by implementing the unifyType:with: method which should return the result of unifying the two types passed as arguments.

Since the non-nil type system only knows a single type, subtyping and unification are absolutely trivial. The nonNil type is a subtype of itself and the nonNil type unified with itself gives the nonNil type again:

```
is: aType subtypeOf: anotherType
 ↑ (self isNilType: aType) and: [self isNilType: anotherType]

unifyType: aType with: anotherType
 ↑ ((self isNilType: aType) and: [self isNilType: anotherType])
  ifTrue: [self singleType]
  ifFalse: [self topType]
```

3.1.6. The built-in static type system

Although Smalltalk is dynamically typed, its source code nevertheless contains some inherent static type information. For example, the class of an object is statically known if it appears as a literal in the source code. Static type information such as this can be very valuable to the type-checking algorithm and to any given pluggable type system. It is therefore important to capture this information and make it available in a convenient form. TypePlug achieves this with a built-in static type system which is itself implemented as an ordinary pluggable type system.

Every expression is assigned one of the following static types:

- The self-type, the type of the pseudo-variable self.
- The super type, the type for the pseudo-variable super.
- Class types, the types for expressions whose class is known.
- Object types, the types for expressions whose exact value is known, e.g. literals and globals.

- Block types, the type for literal blocks.
- The top type, meaning that nothing is known about the static type of that expression.

Object types of a given class are all subtypes of the corresponding class type, e.g. the object type for the integer literal 42 is a subtype of the class type for SmallInteger.

Block types are composites: they describe the types of arguments and a return values of blocks. The block type is a class type, since the class of a literal block is obviously known. The same goes for the super type. The self-type, however, is not a class type—in class hierarchies the pseudo-variable self can refer to any of the subclasses of the class it appears in.

In contrast to user-defined pluggable type systems, the static type system is always present underneath every other type system. The built-in static types are available to implementors of pluggable type systems, and can be exploited when defining subtyping and unification for those systems. As a general principle, whenever an implementation gets ahold of a type of some expression (e.g. the two types passed as arguments into the unification operation) it can also access the static type of that same expression.

3.2. Type checking

The heart of TypePlug is its approach to type-checking. The type-checking algorithm has been designed to make it easy to plug in a new type system. Furthermore, since type annotations in pluggable type systems are optional, the algorithm must deliver useful results in the face of an only partly annotated codebase. As a consequence of these considerations we established two guiding principles for the approach to type checking:

(1) Only code that contains type annotations or uses annotated methods will be type-checked.
(2) Where code to be type-checked refers to unannotated code, static types and type inference will be used.

The combination of these two principles makes it possible to deal well with a mix of typed and untyped code.

3.2.1. Ensuring type safety

Type checking in TypePlug is applied per method and per type system. The type checker takes a type system and a method as input, statically checks the method for type safety and returns detailed results about a type error if there is one.

Source code is analyzed by traversing its AST representation. A type is assigned to every expression node in the AST. The type of an expression is determined by the typing methods of the type system and—where possible—by type inference, taking into account the type and cast annotations in the source.

While the type checker traverses the AST, at certain points type safety is ensured by doing subtyping checks. The three points are assignments, return statements and message sends. Type safety checks for assignments and return statements are trivial, but message sends deserve an extended discussion.

3.2.2. Checking message sends

Ensuring the type safety of message sends is the most difficult problem in type checking Smalltalk. In general, the class of a receiver of a message is not statically known, so there is usually no way to statically determine which method will be invoked at runtime or if a matching method even exists. Clearly, any static type-checking algorithm relying on partial type annotations must be pragmatic here and make some compromises.

One possible solution to improve the situation is to extensively use type inference to determine the class of message receivers. We do utilize some simple forms of type inference but in the context of a specific pluggable type system, not to determine the class of expressions.

We use a simple scheme that nevertheless yields useful results. Our approach tries to look up the *set of methods* that could be invoked for a specific message send and involve the whole set during type checking. The static type system is used to make this set as small as possible. Thus our approach is similar to standard type inference techniques [13], but with a far simpler reduction strategy.

Three cases are distinguished based on the static type of the receiver:

(1) If the receiver class is statically known, i.e. the receiver has a static class type, the method invoked at runtime can be looked up precisely and the set consists of that one method. Note that object types, block types and super types all fall under this case since they are class types.
(2) If the receiver has a self-type we need to consider the class the method being type-checked belongs to. The set consists of all implementors of the message in this class and its subclasses. It is necessary to include subclasses because the self pseudo-variable can refer to an instance of a subclass if the method is called from a subclass.
(3) If the receiver does not have a static type, i.e. its static type is the top type, we have to resort to a very broad strategy: the set consists of all implementors of a message, i.e. all methods of the message's name implemented in the whole system. This case is the most common, unfortunately.

A pluggable type system might carry information that can be used to further reduce the set of methods in this case. That is why type systems get the chance to implement their own strategy for this third case (by overriding the methodsForMessage method).

Once the set of methods to be considered has been fixed, the actual type check of the message send consists of asserting that the types of the arguments are subtypes of the respective types of the methods parameters. If a method parameter does not have a type it is assumed to have the top type which means that untyped parameters effectively are not type-checked. If the set of methods is empty a type error is raised.

This approach as a whole has the desirable property of catching most type errors. But it has the undesirable property of possibly raising too many type errors. When unrelated classes have methods with the same name but with different parameter types, the "all implementors" strategy might label a message send as a type error even though at runtime the error would never occur. This drawback becomes less problematic when one considers what actually happens when type checking arguments in a message send. If a method does not have any type annotations argument type checking is always successful. Only methods that declare types on their arguments can provoke type errors. Since we do not expect to operate in a fully annotated code base these questionable type errors should not occur often.

One problem remains: This approach cannot guarantee that a receiver actually responds to a message at runtime, i.e. that a receiver actually implements a method of that name. We only guarantee type safety for the case that a receiver actually responds to a message.

3.3. Type inference

We treat type inference as a crucial tool to enhance the user experience of TypePlug. It spares a programmer from exhaustively annotating source code with types.

Our type checker has two specific limited forms of type inference built in, for local variable types and for method return types. Both of those are optional, insofar as the type checker just *tries* to infer types but does not depend on a conclusive result. What is more, the pluggable type systems themselves do not have to care about type inference—it emerges from our model of type systems and does not restrict expressiveness within that model.

Type inference for local variables is not particularly novel or interesting, so we will only discuss our approach to return type inference here.

3.3.1. Return type inference

When the returned result of a message send is used, the type checker needs to make a useful assumption about the return type of the message. Again, Smalltalk's dynamic properties make it impossible to know which method will actually be invoked at runtime. So we use the same rules as with argument type checking described in Section 3.2 to get a set of methods to be considered. The return type of the message is then the union of all return types of those methods.

If a method was not annotated with a return type, its return type is by default considered to be the top type. But to improve the quality of type checking and comply with the requirement to infer as many types as possible, the type checker will try to infer the return type of an unannotated method. Our simple-minded type inference basically works the same way as type checking (implementation-wise it is in fact identical): we walk the AST of the method and keep track of the types of AST nodes. The inferred return type simply becomes the union of the types of all return statements in the method. If the method's last statement is not a return statement, the type defined by the type system for self is also part of this union since by default Smalltalk methods return self.

When inferring the return type of a method the result can obviously be improved by inferring the return types of message sends within that method as well. In fact the inferencer could drill even deeper and walk the whole graph of method calls to make the result as precise as possible. With our simple inference strategy, this is not realistic for performance reasons, so we limit the inferencer to an *inference depth*. It defines how deep the inferencer looks into the call graph.

Increasing the inference depth is very costly in terms of performance, since the total number of methods considered grows very fast. While using TypePlug with real world code we discovered that 3 is about the highest tolerable inference depth. We use various caching strategies to improve performance, but in general inferred return types cannot be kept in a cache very long because the conditions that lead to a cache invalidation are very costly to detect, especially with a high inference depth (i.e. when the return type annotation of a method changes, the inferred return types of all its callers might change).

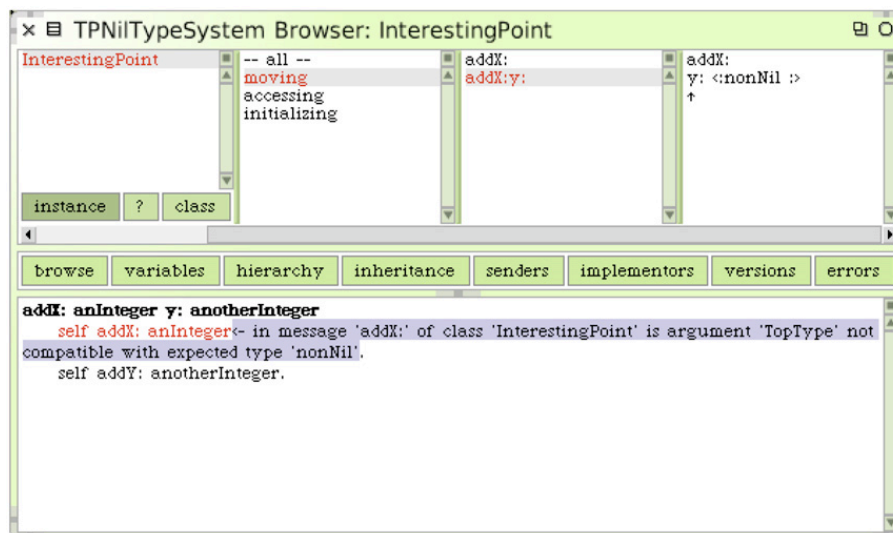### 3.3.2. The impact of return type inference

To assess the impact of return type inference, we carried out the following experiment. In a stock Squeak image without any explicit type annotations we tried to infer the return type of all methods of classes in the category Files-Directories. This category contains Squeak's abstraction of file system directories, with a total of eight classes with 210 methods. This serves as an example of a typical small package of user code that one would want to type check.

In the second stage of the experiment, we added as many return type annotations to methods of the Object class as possible. Object is the base class for almost all other classes in Smalltalk; it implements some heavily used methods such as = (equality), class and copy, so annotating those should improve return type inference (because overridden methods inherit types from su-

Table 2
Return type inference for class category Files-Directories in a stock Squeak image, version 3.9 final

	Non-nil	Class-based
Total methods	210	210
Inferred, without annotations	67	66
In percent	31.9	31.4
Inferred, with annotations	73	77
In percent	34.8	36.7

Fig. 1. The type browser.

perclasses). With this small set of type annotations we again tried to infer all return types in the Files-Directories class category. We used an inference depth of 1, meaning that the type inferencer is allowed to dive one level deep into messages used in an examined method, but not further. We ran this experiment for both the non-nil and the class-based type system (discussed in Section 4.1). The results are summarized in Table 2.

In both type systems, for about 31% of all methods the return type could be inferred even without any type annotations present in the image. This shows that return type inference does add significant value to the system as a whole, since in this package for more than 30% of methods explicit return type annotations are not even needed. Thirty-nine methods (19% of all methods) do not contain a return statement, so their return type is trivially inferred to be the type of self.

With type annotations for Object, inferred return types are about 3% more for the non-nil type system and 5% more for the class-based type system. This improvement is quite impressive considering that only a few annotations were added to Object. These results suggest that the cumulative effect of additional annotations to other commonly used classes such as numbers, strings and collections should be good.

3.4. Programming environment integration

In Smalltalk IDEs, browsers are traditionally used to navigate and modify the source code in an image. Part of TypePlug is a type browser for Squeak, a browser that enhances a standard browser with some type-specific behavior. The type browser has three main features: it integrates type checking, it provides an alternative way of introducing type annotations without changing source code (external type annotations) and it exports types to a distributable form.

Fig. 1 shows the type browser in action. Compared to a standard browser, the type browser has an extra panel to the right of the list of methods. This panel shows the method type of the currently selected method, i.e. its argument and an up arrow ($\uparrow$) to represent the return statement, including types if there are type annotations. Selecting one of these types brings up the type in the bottom (source code) area of the browser where it can be edited.

3.4.1. External type annotations

For a type system to support useful type checking it is often necessary to annotate at least a minimal set of methods of the standard Smalltalk classes with types. For example, you may want to annotate the return type of the hash method of Object with a nonNil type. But normally it is not a good idea to add such an annotation by modifying the source of a method. Redefining methods in standard classes such as Object works well locally in an image, but problems arise when the code or the types should be packaged up for distribution, reside in a version control system, or undergo any other form of migration between images.

To address these issues, the type browser offers a way to add *external type annotations* which are separate from the source code of a method. External types can simply be added and modified through the extra types panel of the browser, which registers types with an external cache but does not change the source code of a method. Additionally, the type browser offers the option to export the types of all methods of a class, including both external and in-source types. Exported types of several classes can easily be bundled, distributed and imported into any image. External type annotations are taken into account for doing type checking and inferencing just like type annotations that are part of the code.

The external type annotation support of the type browser is an important tool for the development and use of pluggable type systems. It allows developers of a type system to distribute a set of types for a Squeak base image, but it also allows other parties to create and distribute type packages for arbitrary type systems.

4. Case studies

In addition to the non-nil type system presented in Section 2, we used TypePlug to implement a class-based type system and a confinement type system. The description focuses on how to implement type systems using TypePlug, we do not provide formal definitions for the type systems used as case studies.

4.1. A class-based type system

We use TypePlug to implement an expressive class-based type system, sporting many features of modern statically typed languages. It supports generic types, polymorphic methods, type unions and typed blocks. The syntax for type annotations in this system is summarized in Table 3. This is expressive enough to meaningfully annotate most Smalltalk code with class-based types.

The type of instances of a class is simply the name of that class, e.g. booleans and integers have the types Boolean and Integer, respectively. Classes themselves (as opposed to their instances) have a type, too. Such a class type is formed by appending class to the name of the class, e.g. Boolean class for the type of the class Boolean.

We will not discuss all other kinds of types in detail. Instead we will look closely at the important subtyping relation and the polymorphism features of the type system.

4.1.1. Subtyping

The usefulness of a class-based type system such as this one depends largely on the definition of subtyping. Smalltalk code in general is not well suited to be typed using classes. For example, classes might override methods while breaking assumptions about types made in the superclass, and other classes might "delete" methods defined in superclasses. All of these things function well considering Smalltalk's dynamic properties, but are not readily compatible with the notion of subtyping.

On the basis of these considerations we define a subtyping relation that is based on the *type interface* of a class. We define this to be the set of class method types that have some type annotation (a method type is a method's name with its argument and return types). Fig. 2 demonstrates this with an example. The type interface of the type Fruit is the method type of mixWith: (with argument Fruit and return type Array E: Fruit, which is a generic type. We discuss generic types in Section 4.1.2) plus of course any method types inherited from superclasses. isVegetable is not part of the type interface since it does not have any type annotations.

The definition of our subtyping relation consists of two clauses. The first clause forms the basis for the relation with a standard contravariant definition for structural subtyping:

1. Type A is a subtype of type B if the type interface of B is a subset of the type interface of A and for all method types mB of the interface B and the corresponding method type mA of the interface A it is true that: the argument types of mB are subtypes of the respective argument types of mA and the return type of mA is a subtype of the return type of mB.

Taken in isolation, this is a standard subtyping relation from the literature (e.g. [14, p. 182]) based on structural interfaces, using the usual contravariant subtyping rule for arguments.

Table 3
Grammar for class-based types

Class :: =		
	ClassName	
	Self	
	Instance	
Type :: =		
	Class	(Simple type)
	Class class	(Class-side type)
	Type | *Type*	(Union type)
	Block args: {*Type**} return: *Type*	(Block type)
	Class (*ParamName*: *Type*)+	(Generic type)
	Param *ParamName*	(Type parameter)
	MethodParam *ParamName*	(Type parameter)

```
class Fruit subclassing: Object
    mixWith: aFruit <:type: Fruit :>
        ↑ (Array with: self with: aFruit) <:type: Array E: Fruit :>
    isVegetable
        ↑ false

class Apple subclassing: Fruit
    color
        ↑ (Color red) <:type: Color :>

class Orange subclassing: Fruit
    color
        ↑ (Color orange) <:type: Color :>
```

Fig. 2. Fruit bowl code.

According to this first clause, in the fruit example from Fig. 2 it is clear that both types Apple and Orange are subtypes of the type Fruit; their type interface contains the (inherited) mixWith: method with types. Also, more surprisingly, the Apple and Orange types are both subtypes of each other—their type interfaces are identical since they both implement an annotated color method.

There is one obvious problem with subtyping based only on the above first clause: if this were the complete subtyping rule then every simple type would be a subtype of every other simple type in a codebase without any type annotations. This would not be useful, which motivates the second clause of our definition:

2. We consider the type B of the class B which is a subclass of A. If the type interface of B is identical with the type interface of A (i.e. the methods of class B do not add or change any type annotations compared to A), then the type B has no subtypes except itself.

This means that for types based on classes that do not define any typed methods, subtyping is based on type identity only. As a consequence of this rule, if Apple and Orange introduced no type annotations of their own, then they would be considered to be distinct types, unrelated to each other. This rule prevents unannotated classes from collapsing to a common type.

This rather unusual subtyping definition has some interesting implications. First, thanks to the second clause, it works well with untyped classes. Second, it puts a lot of responsibility into the hands of the programmer who makes type annotations. For example, a newly created class is not a subtype of its superclass. The programmer must add at least one method with a type annotation to the new class (while not violating subtyping rules) to make the new class a subtype of its superclass. In general, to get subtyping relations in a class hierarchy, type annotations must be added to all the classes in the hierarchy that should participate in the subtyping relation.

4.1.2. Genericity

Generic types are types parameterized by at least one named type parameter. A generic type parameter consists of a name followed by a colon and the type it should be bound to. A typical example of generic types are collections, e.g. the type of an OrderedCollection. This type has a single parameter E referring to the type of the elements of the collection. An OrderedCollection of integers then has the type OrderedCollection E: Integer.

In the context of a class with a generic type, type parameters can be used just like any other type: the type Param E refers to the type that the parameter E takes in a generic type. By using a type parameter in any method a type implicitly becomes generic, i.e. there is no explicit declaration of type parameters. For example, the type of an OrderedCollection is generic because the parameter Param E appears in the methods of the class OrderedCollection, e.g. as the argument type of the add: method.

Apart from type parameters with class scope as used in generic types, our class-based type system also supports type parameters with method scope. Methods making use of such type parameters are *polymorphic methods*. The quintessential and most simple example of a polymorphic method is the id: method which takes an argument and does nothing but return that argument again. To type id: we can give its argument the type MethodParam A, a type parameter named A with method scope. As the return type of id: we use the same type parameter again, MethodParam A. This expresses that we expect the return type of id: to be the type of its argument.

During type checking of code invoking a polymorphic message, the type checker infers the value of all type parameters from the type of the concrete arguments passed (in the type system implementation, this is achieved via the hook method transformMethodType:...).

An interesting example of a polymorphic method is ifTrue:ifFalse: in the class Boolean:

```
ifTrue: aBlock < :type: Block args: {} return: MethodParam R :>
ifFalse: anotherBlock< :type: Block args: {} return: MethodParam R :>
 ...
 ↑ (...) < :type: MethodParam R :>
```

This example demonstrates that polymorphic methods are needed to type some crucial innards of Smalltalk and also illustrates block types.

4.2. A confinement type system

As an example very different from the non-nil and class-based type systems we also define a confinement type system. This type system implements a very specific kind of confinement: confined instance variables. References to mutable objects such as a collection are often considered to be private to a class when they are stored in an instance variable; these references should not be shared with other classes. The confinement type system can guarantee that such confined instance variables do not leak from their class and thus are not modified outside their class. In Smalltalk, all instance variables are always private, encapsulated by the instance, but nothing prevents an object from leaking a reference to its private state as the return value of a public method. In a way, this type system expresses an extended form of privateness for instance variables.

The confinement type system demonstrates how TypePlug enables type systems with fairly complex semantics to be implemented succinctly. The implementation consists of only four classes with a total of about 110 lines of code.

This type system exploits TypePlug's type inference in an unusual way: whether a method possibly returns a confined reference is determined only by return type inference, not by explicit return type annotation. Confined type annotations are added only to instance variables. During type checking, type inference automatically takes care of analyzing the flow of these confined references through methods.

A further interesting aspect concerns the *unconfined types* supported by our confinement system. Unconfined annotations are a pragmatic tool for a programmer to signal to the type checker that he knows the annotated expression does not evaluate to a confined value. The confinement type system is conservative in that it considers the result of any message sent to a confined reference to be confined as well. But some methods (such as copy to copy an object) will always return new and thus unconfined references. The return types of such methods can be annotated with an unconfined type. Unconfined types effectively "override" the effect of confined types—this use of overriding could be a pattern useful in other type systems.

4.2.1. Working with confined instance variables

In this section we explain our confinement type system through a usage scenario. As an example we consider a class Directory that defines a file system directory containing a number of files stored in the instance variable files.

```
Object subclass: #Directory
typedInstanceVariables: 'files <:confine: toClass : >'
 ...
```

Since the instance variable files holds a mutable list of files, it better not be modified outside the class. This constraint is expressed by annotating the variable with **<:confine: toClass :>**.

We assume the class has an accessor method files that simply returns the instance variable files. A user of this accessor is the class Archive representing a compressed archive containing several files (e.g. a ZIP file). The class has a method addDirectory: to add all files of a directory to an archive.

```
Directory ≫ files
 ↑ files
Archive ≫ addDirectory: aDirectory
  members
   ifNil: [members : = aDirectory files]
   ifNotNil: [members addAll: aDirectory files]
```

When the type checker is run on the addDirectory: method it complains about the first files message send with the error message "method is returning a confined value of Directory". Through return type inference it discovered that the result of the message send might have a confined type, which is not allowed in a class other than Directory. The potentially damaging side effects of such a leak of a confined instance variable can be observed in this example: adding a directory to a new archive sets its members instance variable to the same collection that contains the directory's files, and later calls to addDirectory: will thus modify both the archive and the first directory added to it. One could argue that the code in addDirectory: is bad style and that members should be initialized to an empty collection anyway—but the type system is meant to detect exactly these kinds of oversights.

Confinement violations such as the one detected here can usually be fixed by operating on copies of the confined values. Here, one would have the files accessor return a copy of the instance variable:

```
Directory ≫ files
    ↑ files copy
```

This eliminates the type error since files now has a neutral return type. The reason why this works is a bit more subtle than it may appear at first sight and needs an explanation about how the return type is determined in this case. By default, the return type of a message sent to a receiver with a confined type is considered to be that same confined type. The only exception are methods where the static type system can prove that the return type is not a static self-type. The rationale behind these rules is that any method could be returning self, a reference to the receiver which of course keeps the receiver's confinement restrictions.

But some methods such as copy inherently always return a new and thus unconfined reference, even though this might not be statically provable. To allow such methods to be tagged accordingly by the user the confinement type system knows about the type unconfined. In our directory example, the copy method's return type was annotated as **<:confine: unconfined :>**. The unconfined return type will always override any confinement restrictions that might have been inferred for a method.

There is another important use case for the unconfined type, in casting. Consider the following method from the Directory class:

```
Directory ≫ copyAllTo: aDirectory
    aDirectory addAll: files<:castConfine: unconfined :>
```

The confinement type system forbids the usage of confined references as arguments in a message send. This is necessary because the type checker has no way to tell whether anything unwanted happens to a confined reference once it is passed to a method. But since there are legitimate and safe uses of confined references as message arguments, they can be explicitly cast to the unconfined type. This is a strategy that many type systems implemented with TypePlug might adopt: better generate too many type errors to be safe, but let the user control these type errors through casts.

4.2.2. Keeping references confined

Usually important logic of a type system is contained in the subtyping relation and unification operation, but here they are rather simple and we will not discuss them. The interesting aspect of this type system is how we make sure that confined references do not leak from a class. In general, references only leave the bounds of a class through message sends, so most violations of "type safety" in this type system happen there. There are two conditions that make a message send unsafe:

(1) The return type of the message send is a confined type associated with a class A not compatible with the current context class B (meaning A and B are not identical or B is not a subclass of A).
(2) One of the arguments has a confined type.

Examples of both of these type errors were given during the scenario in the previous section.

Another kind of message send needs special attention: those where the receiver has a confined type. To be on the safe side, we must assume that a message sent to a confined receiver returns a reference to self, meaning that the return result is confined as well (except if the return type is explicitly unconfined). For some cases, return type inference can statically determine (using the static type system) that the return type of a message send is definitely not self. But unfortunately the framework and hooks into the type checker do not currently offer a way to make decisions based on the inferred return type of a method, so we cannot single out this special case.

There is another potential source of type errors: assignments. When a confined reference is assigned to a variable, the variable would have to become confined if it is not already. But our type checking algorithm does not offer the option to change the type of an instance variable during type checking. So we explicitly forbid assignments where a confined reference is assigned to an untyped instance variable, class variable or global variable. These restrictions on message sends and assignments are implemented within the transformMethodType:… and assignmentTo:… hooks, raising custom type errors. In the case of assignments, the error message will suggest that the left-hand side variable be declared as confined as well.

4.2.3. Drawbacks of the approach

The type system works remarkably well in confining instance variables, partly by being conservative (e.g. considering the result of messages sent to a confined reference to be confined). But there is one caveat: this confinement type system relies on the type inference capabilities of TypePlug. Whether a method returns a confined reference or not is determined solely by inference of return types, not by explicit type annotations as in, e.g. the non-nil type system. But return type inference is mostly just a tool to enhance the quality of type checking; it is not supposed to (and, in fact, cannot) work for all methods.

In this type system, problems arise if a confined reference enters a method through multiple levels of indirection. Return type inference only walks the tree of message sends to a fixed inference depth—if some method past that depth returns a confined reference that reference is not considered during return type inference. Here is an example to illustrate this point.

```
Directory ≫ files
  ↑ files
Directory ≫ firstFiles: anInteger
  ↑ (self files size >anInteger)
    ifTrue: [self files first: anInteger]
    ifFalse: [self files]
Device ≫ directorySneakPeeks
  ↑ directories collect: [:dir | dir firstFiles: 3]
```

Provided the instance variable files of Directory is confined, it is leaking from Directory ≫ firstFiles: and thus violates confinement in Device≫directorySneakPeeks. But if we are type checking directorySneakPeeks with an inference depth of just one no type error is detected, since the type checker will not try to infer the return types of methods within firstFiles: and not find out that the files method returns a confined reference.

The way to minimize missed confinement violations due to this problem is obviously to use a large inference depth. Increasing the inference depth does however slow down type checking considerably. With the current state of TypePlug a depth of about 3 is empirically the highest tolerable value. We recommend to develop using a low value and occasionally let the type checker run on all methods in relevant packages with a higher inference depth (some support for this is provided with the TPImageChecker class). This occasional exhaustive type checking could for example be part of a build process.

There is one other caveat: the confinement in this type system is based on classes, not on instances. This means that it cannot prevent two separate instances of a class from sharing confined references. There is not much that can be done about this, since bounds between instances cannot generally be determined statically. Runtime checks, complementing TypePlug's static type checking, would be necessary to implement instance-based confinement.

5. Discussion and related work

5.1. Reflection and exceptions

In a dynamically typed language like Smalltalk there are programming idioms that our type checker simply is not capable of handling. We discuss two problems of our approach: typing code that uses the reflective features of Smalltalk and exception handling.

Reflection: Static type checking is about deducing static properties of a system before it is run, whereas reflection (not just introspection) means changing the structure and behavior of a system at runtime. Typing reflective systems thus poses many problems [15].

The perform: method is used to send a message to an object with the message name determined at runtime. It takes a message name as its argument and sends that message to the receiver. The return type depends on the value of the argument which in general is not known statically, so there is no way to reasonably type check a perform: message, its arguments and its return value. However, in many cases we can add an explicit type cast to a perform expression, thus providing the type checker with the information needed to check a perform expression.

The doesNotUnderstand: idiom poses a more severe problem for our type checking algorithm. The doesNotUnderstand: method allows classes to intercept runtime errors occurring when a message was sent that is not understood by the receiver. This means any object possibly accepts a message sent to it even though its class does not explicitly implement a method of the same name. This is a problem because type checking by default assumes that it can find all implementors of a given message name. To avoid false positives when type-checking, the only recommendation we can make is to avoid this idiom in areas of code that should be type safe.

Exceptions: TypePlug currently does not model exceptions. In general, every message send in Smalltalk could result in an exception being signaled. The concept of exceptions is fortunately orthogonal to our notion of type safety. For our purposes, if an exception is signaled it simply means that the execution of a method stops at that point; it does not in any way affect the types of variables or other expressions. TypePlug simply does not deal with exceptions at the moment, not diminishing the usefulness of type checking.

5.2. Related work

In the following, we give an overview of related work. After a short presentation on relevant general work on type systems and type inference in the context of Smalltalk, we focus on related work on pluggable type systems.

5.2.1. Type systems and type inference for Smalltalk

Smalltalk as a practical dynamically typed system has seen many proposals for adding static type systems [16–19]. The most recent effort at conceiving and implementing a practical type system for Smalltalk was *Strongtalk* [20]. Some of these type systems support optional typing, but they provide just a single type system, not addressing pluggability.

Type inference was originally researched in the context of functional languages [21]. Type inference then was applied to Smalltalk [22]. Palsberg and Schwartzbach presented the first algorithm that can type check completely untyped Smalltalk code [13]. The Cartesian product algorithm (CPA) by Agesen [23] provides a substantial improvement in both precision and efficiency. Even with this advanced algorithm, type inference does not scale to larger programs. Efficiency and scalability thus is a focus of current research. Demand-driven type inference (DDP) [24] provides scalability by analyzing type information on demand and selectively reducing precision. RoelTyper [25] uses heuristics for providing type information of instance variables. RoelTyper, like DDP, provides high performance at the cost of reduced precision.

5.2.2. Pluggable type systems

Pluggable type systems were originally proposed by Bracha [1]. A number of implementations of pluggable type systems have been published.

JavaCOP [7] is a program constraint system for implementing practical pluggable type systems for Java. The authors present a framework that allows additional types to be added to Java source code, based on Java's annotation facilities. They then define a declarative, rule-based language that can express rules as constraints on AST nodes, making use of the information from annotations as well as from Java's static type system. These rules form the semantics of pluggable type systems and can be enforced by a type checker that hooks into the compile process.

Annotations in JavaCOP are similar to TypePlug's, but are restricted to class and variable declarations by the Java language, so something like our casts on arbitrary expressions cannot be supported. The way type systems are implemented with rules in a domain-specific language is very different from TypePlug's type system model. The rule language enables very fine grained control over type checking. The authors prove the validity and versatility of this approach by implementing an impressive number of pluggable type systems, ranging from confined types to reference immutability and checks of the kind usually done by coding style analyzers. On the other hand, our approach with a rather fixed type checking algorithm and customization is arguably a bit simpler and results in simpler implementations at least for some applications. For example, our non-nil type system is extremely simple with about 30 (mostly very trivial) lines of code while JavaCOP's equivalent presumably needs several non-trivial rules. JavaCOP does not provide any special support for working with untyped legacy code. We consider simplicity of implementation to be important in order to be able to experiment with different kinds of type systems and typing strategies.

Fleece [6] also explores the notion of pluggable type systems, but explicitly for dynamically typed languages. As an example of a dynamically typed language the report introduces and defines $\mathscr{R}_{sub}$, a subset of the Ruby programming language, which is then used throughout the report. It develops the notion of annotators that automatically add and propagate annotations (types) on nodes of the AST of a program. A special case of an annotator is the programmer who can manually add annotations to a program. Fleece's handling of annotations correspond in many aspects to TypePlug's. Automatic annotators play the role of the inference part of TypePlug's type checking.

Compared to TypePlug, Fleece is both more general and more restricted. It is more general, because it is independent of the language $\mathscr{R}_{sub}$; it copes with any other language that can be expressed in the grammar form that it expects. However, it is more restricted because it has not been shown to handle a real dynamically typed language. $\mathscr{R}_{sub}$ is a very restricted form of Ruby with, e.g. no inheritance or support of the standard library. The feature that distinguishes TypePlug is the ability to pragmatically work with an actual full implementation of Smalltalk, thus supporting type-checking of legacy code.

JastAdd, an extensible Java Compiler, has been extended with a pluggable type system by Ekman and Hedin [5]. The paper presents only one type system, a non-nil type system similar to that of TypePlug . Of all three discussed other frameworks, it is the only one which provides support for reducing the number of annotations needed when dealing with untyped legacy code. It supports inference of non-null types, but it does not allow for type annotations without changing the source code.

6. Conclusion and future work

TypePlug is intended to offer a very pragmatic framework for implementing pluggable type systems. New type systems are defined by specializing the TPTypeSystem class and overriding the methods for subtyping and type unification. TypePlug makes it relatively easy to obtain the benefits of a pluggable type system even if the underlying legacy code base remains largely free of type annotations. This is achieved by (i) only type-checking annotated code, (ii) using static type information and limited type inference to reason about unannotated code, and (iii) externally annotating third party code. Explicit type casts can also be used to avoid annotating code that is known to be safe.

We have made experience with three very different kinds of type systems, including a classical class-based type system, a non-nil type systems and a type system for confined types.

There are numerous further directions to explore. Presently TypePlug does not take exceptions into account. Type inferencing is quite slow, and inferencing beyond a depth of three levels is impractical. We plan to integrate modern type inference algorithms and a heuristical type analysis system in the spirit of RoelTyper.

An interesting field for future research is to explore type checking in the presence of reflection, especially how to type check a system that can be changed reflectively at runtime. It would also be interesting to explore ways of adding optional run-time type checks by making further use of the Persephone framework for sub-method reflection, thus complementing TypePlug 's static type checking. Pluggable type systems might be able to benefit from information provided by any other pluggable type system, rather than just the built-in static type system. This sharing of type information is another direction to consider.

Acknowledgements

References

[1] Bracha G. Pluggable type systems. In: OOPSLA workshop on revival of dynamic languages, October 2004. URL: ⟨http://pico.vub.ac.be/%7Ewdmeuter/RDL04/papers/Bracha.pdf⟩.

[2] Grothoff C, Palsberg J, Vitek J. Encapsulating objects with confined types. In: OOPSLA '01: proceedings of the 16th ACM SIGPLAN conference on object oriented programming, systems, languages, and applications. New York, NY, USA: ACM Press; 2001. p. 241–55. URL: ⟨http://www.cs.ucla.edu/~palsberg/paper/oopsla01.pdf⟩.

[3] Smith F, Walker D, Morrisett JG. Alias types. In: ESOP '00: proceedings of the 9th European symposium on programming languages and systems. London, UK: Springer; 2000. p. 366–81 URL: ⟨http://www.it-c.dk/people/birkedal/teaching/rar-seminar-Fall-2001/papers/alias.ps.gz⟩.

[4] Zhao T, Noble J, Vitek J. Scoped types for real-time Java. In: RTSS '04: proceedings of the 25th IEEE international real-time systems symposium (RTSS'04). Washington, DC, USA: IEEE Computer Society; 2004. p. 241–51 URL: ⟨http://jiangxi.cs.uwm.edu/publication/drafts/scoped04.pdf http://jiangxi.cs.uwm.edu/publication/rtss04.pdf⟩.

[5] Ekman T, Hedin G. Pluggable checking and inferencing of non-null types for Java. Journal of Object Technology 2007;6(9):455–75.

[6] Allwood T. Fleece, pluggable type checking for dynamic programming languages. Master's thesis, Imperial College of Science, Technology and Medicine, University of London. URL: ⟨http://wwwhomes.doc.ic.ac.uk/~tora/previous/project/Report.pdf⟩.

[7] Andreae C, Noble J, Markstrum S, Millstein T. A framework for implementing pluggable type systems. In: OOPSLA '06: proceedings of the 21st annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications. New York, NY, USA: ACM Press; 2006. p. 57–74.

[8] Haldiman N, Denker M, Nierstrasz O. Practical, pluggable types. In: Proceedings of the 2007 international conference on dynamic languages (ICDL 2007). ACM Digital Library; 2007. p. 183–204 URL: ⟨http://www.iam.unibe.ch/~scg/Archive/Papers/Hald07b-Typeplug.pdf⟩.

[9] Black A, Ducasse S, Nierstrasz O, Pollet D, Cassou D, Denker M. Squeak by example, Square Bracket Associates, 2007 ⟨http://SqueakByExample.org/⟩. URL: ⟨http://SqueakByExample.org/index.html⟩.

[10] Ducasse S, Nierstrasz O, Schärli N, Wuyts R, Black A. Traits: a mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) 2006;28(2):331–88 URL: ⟨http://www.iam.unibe.ch/~scg/Archive/Papers/Duca06bTOPLASTraits.pdf⟩.

[11] Denker M, Ducasse S, Lienhard A, Marschall P. Sub-method reflection. In: proceedings of TOOLS Europe 2007, vol. 6. ETH; 2007. p. 231–51. URL: ⟨http://www.jot.fm/issues/issue_2007_10/paper14 http://www.jot.fm/issues/issue_2007_10/paper14.pdf⟩.

[12] Marschall P. Persephone: taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006. URL: ⟨http://www.iam.unibe.ch/~scg/Archive/Diploma/Mars06a.pdf⟩.

[13] Palsberg J, Schwartzbach MI. Object-oriented type inference. In: Proceedings of OOPSLA '91, ACM SIGPLAN notices, vol. 26, 1991. p. 146–61. URL: ⟨http://www.cs.purdue.edu/homes/palsberg/publications.html⟩.

[14] Pierce B. Types and programming languages. Cambridge: The MIT Press; 2002 URL: ⟨http://www.cis.upenn.edu/~bcpierce/tapl/index.html⟩.

[15] Alanko LE. Types and reflection. PhD thesis, University of Helsinki, November 2004.

[16] Borning AH, Ingalls DH. A type declaration and inference system for Smalltalk. In: Proceedings of POPL '82, Albuquerque, NM, 1982. p. 133–41.

[17] Johnson, RE. Type-checking Smalltalk. In: Proceedings of OOPSLA '86. ACM SIGPLAN notices, vol. 21, 1986. p. 315–21.

[18] Graver J. Type-checking and type-inference for object-oriented programming languages. PhD thesis, University of Illinois at Urbana-Champaign, August 1989.

[19] Palsberg J, Schwartzbach MI. Object-oriented type systems. New York: Wiley; 1993.

[20] Bracha G, Griswold D. Strongtalk: typechecking Smalltalk in a production environment. In: Proceedings of OOPSLA '93. ACM SIGPLAN notices, vol. 28, 1993. p. 215–30. URL: ⟨http://bracha.org/oopsla93.ps⟩.

[21] Milner R. A theory of type polymorphism in programming. Journal of Computer and System Sciences 1978;17:348–75.

[22] Suzuki N. Inferring types in smalltalk. In: POPL '81: proceedings of the 8th ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, NY, USA: ACM Press; 1981. p. 187–99.

[23] Agesen O. Concrete type inference: delivering object-oriented applications. PhD thesis, Stanford University, December 1996.

[24] Spoon SA, Shivers O. Demand-driven type inference with subgoal pruning: trading precision for scalability. In: Proceedings of ECOOP'04. 2004. p. 51–74.

[25] Wuyts R. RoelTyper, a fast type reconstructor for Smalltalk, 2005. URL: ⟨http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/⟩.