11-23-1994

Foundations of Object-Oriented Languages

Giuseppe Castagna
Liens-DMI

Gary T. Leavens
Iowa State University

Foundations of Object-Oriented Languages

TR94-22
Giuseppe Castagna and Gary T. Leavens

November 23, 1994

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

Foundations of Object-Oriented Languages

2nd Workshop report

Giuseppe Castagna and Gary T. Leavens

TR #94-22
November 1994

Keywords: object-oriented programming, type checking, verification, modules, binary method, matching, subtyping, record update, state, semantics, covariance, contravariance, flow analysis, self, method schemas.

1994 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented Programming; D.2.2 [*Software Engineering*] Tools and Techniques — modules and interfaces; D.2.2 [*Software Engineering*] Program Verification — correctness proofs; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types, modules, packages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — logics of programs; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — algebraic approaches to semantics, denotational semantics, operational semantics; F.3.2 [*Logics and Meanings of Programs*] Studies of Program Constructs — type structure.

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Foundations of Object-Oriented Languages

2nd Workshop report

Giuseppe Castagna* Gary T. Leavens†

November 23, 1994

Abstract

A report on the workshop *Foundations of Object-Oriented Languages*, Paris, July 1994.

1 Introduction

In Paris on July 1–2 1994, researchers gathered at the second workshop of a series of NSF and ESPRIT-sponsored workshops to discuss the foundations of object-oriented (OO) programming languages. This series of workshops is organized by Kim Bruce and Giuseppe Longo. The previous edition was held in Stanford and has been reported in [7].

This year's workshop was held in the Conservatoire National des Arts et Métiers and organized by the Laboratoire d'Informatique de l'Ecole Normale Supérieure; local arrangements were handled by Roberto Bellucci and Giuseppe Castagna. Participation was by invitation only (see the appendix).

The purpose of this workshop was to exchange ideas about the latest research in the semantics of OO programming languages. The workshop consisted of several presentations and some discussion. A lively email discussion also developed shortly after the workshop about binary methods.

This report is organized as follows. Section 2 summarizes the talks given at the workshop. For each talk, one or two references to the work are given. (The bibliography and document pointers are available on the World-Wide Web via the URL: `http://www.cs.iastate.edu/~leavens/FOOL94.html`.) Section 3 summarizes the discussions that took place at the workshop, and afterwards.

*LIENS (CNRS), 45 rue d'Ulm, 75005 Paris, FRANCE. Internet: `castagna@dmi.ens.fr`

†Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040 USA. Internet: `leavens@cs.iastate.edu`.

2 Reviews of Presentations

Ten of the participants made presentations. The following are brief summary reviews of their talks. For each talk, we try to describe the problem being solved, and then give an overview of the solution and its significance. The summary follows the order in which the presentations were given.

Kim Bruce: *Matching is better than constraining for bounded polymorphism in OOLs.*

Kim Bruce discussed his work on providing a flexible, yet type-safe, polymorphic and imperative OO programming language [9]. In an imperative OO language, there are fewer subtypes than in previously-studied theoretical OO languages that do not have mutation. Hence bounded polymorphism in which the bounding relationship is a subtype relationship is not as useful as might be desired. This problem underlies the problem users experience with contravariance rules, and is one reason Eiffel uses an unsafe covariant rule.

PolyTOIL, is a polymorphic, imperative OO language that is type-safe. PolyTOIL has expressions that denote classes, and thus a class has a type, written `ClassType(`σ`,` τ`)`. This type is distinguished from the type of its objects, `ObjectType` τ, because the object type suppresses information about the instance variables (σ).

An important property of PolyTOIL is that methods do not have to be type-checked each time that they are inherited. This is accomplished by giving `self` the type `MyType`, and only assuming that `MyType` "matches" `ObjectType` τ. The definition of *matches* implies that if class C' inherits from C, and the corresponding object types are `ObjectType` τ' and `ObjectType` τ, then `ObjectType` τ' matches `ObjectType` τ. Using matching permits PolyTOIL to have a type system that is nearly as expressive as Eiffel (which uses unsafe covariant type-checking), but which is provably type safe, and does not have Eiffel's link-time global check. This expressiveness is especially important for methods that take or return

an argument of type `MyType`.

Bruce contrasted matching with subtyping, and noted that in PolyTOIL these orderings on types are distinct. He noted that matching is more useful than subtyping as the ordering in bounded polymorphism. For example, consider a polymorphic function that needs a `comparable` type argument, T, where `comparable` is a type with a binary method `leq` of type `MyType` $\rightarrow$ `Bool`. This works better if the bound is "T matches `comparable`" rather than requiring that "T is a subtype of `comparable`."

Benjamin Pierce: *Positive Subtyping.*

Benjamin Pierce described joint work with Martin Hoffman [15]. The problem he addressed is the record update problem: how to type and reason about update in a λ-calculus model of an OO language without mutation. An update function is one that returns an object of the same type. For example, consider a type `Point`. A function `bump` of type `Point` $\rightarrow$ `Point` is an update function. The famous subclass of `Point`, `ColoredPoint` would also have a `bump` function, but this one of type `ColoredPoint` $\rightarrow$ `ColoredPoint`. The problem is how to model `bump`, with a polymorphic function; as is well known, the usual bounded polymorphic definition does not work.

Several complex solutions for this problem have been offered by others. Pierce offered a much simpler solution to a smaller problem: that is a solution in which subtyping for function types is only applied in positive positions, such as function result types. With this restriction, the semantics of a subtype relationship, $S \leq T$, is not just a coercion function from S to T, but a pair of functions. The first element of the pair is the coercion from S to T, while the second function, of type $S \rightarrow T \rightarrow S$, is able to update the T part of an S object.

In the second part of his talk, Pierce discussed equational laws for reasoning about the calculus, and an example proof. The program that is the subject of the proof contains two class definitions, the second inheriting some of its behavior from the first. Although both classes involve recursive self-reference through the pseudo-variable `self`, the proof is structured so that properties of the second class are established without looking at the implementation of the first class. Thus the proof is modular in the same sense that the object-oriented program is modular.

Didier Rémy: *Programming objects in ML-ART.*

Didier Rémy described a solution to the problem of supporting OO programming idioms in ML [20]. The idea is to use the OO model proposed by Pierce and Turner [18] in a type inference framework. To that end he enriched the ML+references language by some features already present in the type inference literature. More precisely ML+references is extended by extensible records (to obtain polymorphic extension for inheritance), projective types (to describe the type structure of fields independently of their presence), existential types (to have state encapsulation) and recursive types.

The difficult problem in ML is to get the objects to share the vector without exposing the representation to clients of the object. To add inheritance, one must support `self`. The usual encoding of object records with `self` is recursive, so one would have to create objects with an unsafe fixpoint, which would not work with the ML type system. In order to avoid this, Rémy modified the Pierce and Turner model by passing both the with exposed state and `self` to each method, rather than just the exposed state. State encapsulation is still realized though the use of existential types. Recursive types are now required since sending messages create fix points. Inheritance is then coded using wrappers.

The final result is the encoding of an object-oriented style in the extended ML, which allows to write type safe object-oriented programs in a type inference framework. Yet some problems persist in this approach: coercions are difficult to write and must be explicit, message sending cannot be polymorphic (so there is no subtype polymorphism), and creation of recursive values is too limited.

Scott Smith: *State in Object-Oriented Programming Languages.*

Scott Smith discussed joint work with Jonathan Eifrig, Valery Trifonov, and Amy Zwarico [14]. This work addresses the problem of how to provide a type system for an OO language with mutable state. This work also shares many of the goals of Kim Bruce's, in that it also deals with a polymorphic, imperative OO programming language.

Smith discussed an approach to this problem in which the semantics of the imperative OO language are given by translation to a non-OO language. For its static type system heavy use is made of F-bounded polymorphism. He noted that a goal is to find a translation into a typed calculus such that one can refer to `self` in expressions that initialize instance variables. (This allows object-overridable methods, as in Modula-3, and circular structures.)

Smith started by discussing an untyped language, LOOP. In LOOP one can create objects, send messages, and read and write instance variables. To allow one to refer to `self` in expressions that initialize instance variables, class and object constructions have

the following form.

```
c = class(self)
  instance variables x = ... self ...
  methods m = ... self ...
end
o = new c
```

These are translated into the following typed calculus, called SOOP.

$$c = \lambda s.\ \textbf{let}\ y = \{x = \textbf{ref}\ ..self()..,\ m = ..self()..\}\ \textbf{in}\ \lambda().y$$
$$o = \textbf{let}\ r = \textbf{ref}\ \lambda().\bot\ \textbf{in}\ r := c(\lambda().!r()); !r$$

Since there are no reference types in an object, one gets subtyping, but this construction does not catch circular definitions. In discussions it was agreed that Smith's use of F-bounded subtyping is able to do what Kim Bruce's did with matching, but without an explicit matching relation.

Smith gave some rules for reasoning about LOOP. He also discussed the proof of the soundness of the type system of SOOP, which is done by subject-reduction.

Martín Abadi: *An imperative object calculus.*

Martín Abadi presented joint work with Luca Cardelli [3] [2]. The problem he addressed is the definition of a primitive calculus to model object-oriented programming. The necessity of such a calculus is supported by the problems met in performing typing and subtyping. In particular, in this work, the modeling and typing of imperative features is studied.

Abadi started by describing an untyped imperative calculus formed by variables and objects with the primitives of method invocation, method override and object cloning. He showed how these constructs suffice to encode both fields and the "imperative λ-calculus" (pure λ-calculus plus assignment). After having described the operational semantics, Abadi defined a first-order typing for the calculus and proved its soundness. Namely, he proved that, starting from a "sound" store, well-typed programs do not go wrong. Abadi concluded this part of his talk by arguing that in an imperative setting some of the difficulties that resulted in the use of sophisticated functional type theories could be avoided; the example he used is the one with movable points, i.e., points with a method that moves the object: while in a functional language such a method must return a new object with the same type as *self*, in the imperative setting it is not necessary to specify any type, since the method works by side-effects.

Starting from the previous type system, Abadi then presented a partial correctness logic. The idea is to write specifications of objects that generalize the previous types and that serve as invariants. The final product is a Hoare logic on programs written in (a modified version of) the untyped imperative calculus described above. Abadi concluded his talk with an example, showing how to use this logic to prove that a given program computed a given value without modifying the store.

Giuseppe Castagna: *Covariance and contravariance: conflict without a cause.*

Castagna presented work that addresses the long-standing problem of covariance vs. contravariance [10]. The so-called *contravariant* rule, used to subtype function types, while assuring type safety, seemed to prevent satisfactory typing of some special cases, notably when binary methods were involved. For this reason in some systems this rule had been replaced by a *covariant* one, but type safety was lost.

Castagna argued that covariance and contravariance appropriately characterize two distinct and independent mechanisms. The so-called contravariance rule correctly captures the *substitutivity*, or subtyping relation (that establishes which sets of codes can replace *in every context* another given set). A covariant relation, instead, characterizes the *specialization* (in OO jargon, the *overriding*) of code (i.e., the definition of new code that replaces the old one *in some particular cases*). Therefore, covariance and contravariance are not opposing views, but distinct concepts that each have their place in object-oriented systems; both can (and must) be type safely integrated in an object-oriented language. He also showed that the independence of the two mechanisms is not characteristic of a particular model but is valid in general, since covariant specialization is present also in record-based models but it is hidden by the incapability of the existing calculi to model multiple dispatching. Thus he showed how it is possible to model multiple dispatch in the record-based models and argued that this improvement naturally allows one to (sub)type in a straightforward (and type safe) way binary or more generally n-ary methods even in these models. The example he gave was the one of `Point` and `ColoredPoint` with an *equal* method (presented also in other talks of this workshop). He demonstrated that if in the *equal* field of a `ColoredPoint` one specifies both the code for an argument of class `Point` and the code for an argument of class `ColoredPoint` and if the code is dynamically selected by multiple dispatching, it is then type safe to have `ColoredPoint`$<$`Point`.

Gary Leavens: *Modules for Multi-methods: Information Hiding and Program Composition.*

Leavens discussed joint work with Craig Chambers [12]. The problem he addressed was one pointed out by William Cook at previous workshops [13]: how to achieve encapsulation and composability for systems of multi-methods. In comparison to abstract data type languages (such as CLU or Ada), and single dispatching object-oriented languages such as Smalltalk, multi-method systems such as CLOS do not have good encapsulation. A method may be written in any part of the program that specializes on objects of a certain class, and thereby that method may gain access to the representation of such objects. When combining independently developed systems of multi-methods, type errors may occur, for which no part of the program is uniquely responsible.

To solve these problems, Leavens described a module mechanism for the language Cecil [11]. This mechanism directly addresses the encapsulation problem, since one can limit access to fields by declaring them `private`. However, since Cecil has subtyping, it becomes possible for a method to be invoked on an object whose type is defined in a module that is not explicitly imported at the point of the method call. This problem is avoided by requiring subtypes to be defined in "extension modules." The methods defined in modules that extend M are implicitly available at run-time in any module that explicitly imports M.

While the module system solves much of the composability problem, there remains the problem of independent extensions of a given module. To solve this problem, Leavens observed that the person who puts both independently-developed extension modules in a program should be responsible for resolving any problems in a "most-extending module." A unique most-extending module must exist for each module, and this condition can be checked quickly at link-time. This seems to provide multi-methods with the same degree of information hiding as standard OO languages, while retaining other advantages of multi-methods.

Jens Palsberg: *A Type System Equivalent to Flow Analysis.*

Jens Palsberg presented a joint work with Patrick O'Keefe [17]. He addressed the problem of finding a type system that accept exactly the same programs as *safety analysis*. Safety analysis is a flow analysis of programs which collects type information and uses this information to accept only *safe* programs (i.e., programs that cannot go wrong). Such analyses have the advantage that they can be applied to *untyped* languages, where the more traditional abstract interpretation needs types for defining abstract domains.

More in detail he considered the untyped lambda calculus with zero and successor together with the type system defined by Amadio and Cardelli [6] and composed by recursive, arrow, top, and bottom types and Int, plus a subtyping relation with subsumption. Palsberg defined the safety analysis of a term as the resolution of a given set of constraints and showed a cubic time algorithm that computes it. Finally, he proved that a term is typable in Amadio-Cardelli's type system if and only if there exists a solution to the safety analysis. As a result he obtained a type inference algorithm for the given type system, thereby solving an open problem.

Luca Cardelli: *Primitive Object Types with Self.*

Luca Cardelli described joint work with Martín Abadi [2], which is an improvement of the work in [1]. The problem he addressed was the definition of a primitive calculus to model object-oriented programming. The necessity of such a calculus is supported by the problems met in performing typing and subtyping. A focus of this work is the type of `self`.

The starting calculus is the untyped calculus presented in [1], while the starting type theory is the second order type theory of [4] with some improvements: a new quantifier *Obj* is used which, unlike the previous quantifier, can "move points", override self-returning methods, and encode classes. Special annotations are used to identify read-write fields (typically instance variables), read-only fields (typically methods), and write-only fields. These annotations also allow to encode arrow types enjoying the usual contravariance/covariance properties. Cardelli also introduced the notion of *pre-method* (not to be confused with the CLOS concept of the same name), which is a function that is later used to construct a method. This definition is useful since class can then be seen as collections of pre-methods and inheritance relation between classes as pre-method reuse. Although very few primitives are used, the resulting calculus is powerful enough to express object types, class types, and method specialization, covering both class-based and delegation-based frameworks. While the delegation-based frameworks are essentially built-in, Cardelli demonstrated the expressibility of class-based frameworks by closely emulating TOOPLE [8].

Emmanuel Waller: *Method Schemas.*

Emmanuel Waller described joint work with Serge Abiteboul and Paris Kanellakis [5]. He addressed the problem of static verification of consistency of method schemas (decidability and complexity of various cases).

A method schema is a simple programming formalism for object-oriented databases with features such as classes, methods, inheritance, name overloading, and late binding. An important problem is to check whether a given method schema can lead to an inconsistency in some interpretation (a method called with arguments for which this method is undefined). When no restrictions are enforced on method schemas, dynamic errors are possible, and the consistency question is undecidable in general. (This is shown using program schemas.) Decidability is obtained for monadic and/or recursion-free method schemas. In particular, consistency of monadic method schemas is shown to be decidable in $O(nc^3)$ time, where n is the size of the method definitions and c is the size of the class hierarchy; also, it is logspace-complete in PTIME, even for monadic, recursion-free schemas. Method signature covariance is shown to simplify the computational complexity of key decidable cases.

The incremental consistency checking of method schemas is a formalization of the database schema evolution problem. A detailed study of some decidable cases is given in [21].

3 Summary of Discussions

At the end of the workshop there was a general discussion. The discussion started with Luca Cardelli offering his (tongue-in-cheek) list of the 10 most important ideas in OO semantics. The list is given below.

1. subsumption (when one object/method can replace another)
2. subsumption
3. update
4. self-value (methods with `self` as a parameter)
5. self types (Modula-3 doesn't have this)
6. don't forget subsumption (subsumption should still work well with self types)
7. method specialization (methods to allow override, pre-methods)
8. classes and inheritance
9. polymorphism
10. still subsumption (subsumption should still work well with polymorphism)

In reply to this, Kim Bruce said he would put "matching" as items 2 (and 10!) on the list. Other suggestions for this list were: implementation efficiency (Benjamin Pierce), compilability (Carl Gunter), separation of classes and types (Scott Smith).

Some suggested adding multi-methods to the list, and Cardelli noted that you need some sort of encapsulation for that (as in O_2 or the work of Chambers and Leavens). A discussion about modularity ensued, and it was noted that the work of Pierce and Turner [19] and Katiyar, Luckham, and Mitchell [16] allows one to operate on two sets of objects, with representation access, using bounded existential quantifiers.

Carl Gunter started a new line of discussion by saying that, as a semanticist, he would like to get the features of OO in a way orthogonal to other features of a language. One might imagine ML with objects, where one could use OO techniques, but still program in other paradigms. Carl also wanted to know what other ideas were subsumed by OO ideas.

Finally, there was a discussion of binary methods, which reflects their importance in several of the talks. This discussion also continued after the workshop, in email. Pierce asserted that binary methods, which cause so many theoretical problems for modularity and type inference, should be banished. He asked whether there were any useful examples where binary methods were really needed. Cardelli offered a "proof" that binary methods are both useless and worse than useless. The basic idea is that binary methods hurt subsumption, so if one defines what would be binary methods as functions (outside of classes), you win. Cardelli concluded that the only really useful binary methods are multi-methods.

In response, Kim Bruce offered a program that showed a situation in which binary methods are useful (and can be type-checked in PolyTOIL). The example constructed ordered lists of items, in which the items can be compared. Although subsumption was lost because of the binary methods, matching allowed the program to perform a useful function. There were various recodings of this example (by Pierce, Bruce, Rémy and Cardelli). Codings without using matching either needed a more complex type system, or required a change in the interface used by clients of the program.

The result of the extended email discussion was a shared understanding of the extra usefulness of matching and binary methods, of the extra complexity needed to model them, and of specific techniques for modeling them. We now understand well how subsumption and matching can coexist. The issue of whether to eliminate one or the other, or neither, is still unresolved. (Several of the participants in this

discussion are planning an extended report on the binary method problem.)

Acknowledgements

Thanks to Kim Bruce, Luca Cardelli, Giuseppe Longo, Jens Palsberg, Benjamin Pierce, Didier Rémy, Scott Smith, and Emmanuel Waller for comments and corrections to drafts of this report.

Appendix: Participants

The participants in the workshop are listed in Table 1.

References

[1] M. Abadi and L. Cardelli. A theory of primitive objects: second-order systems. In D. Sannella, editor, *Proc. of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 1–25, New York, NY, 1994. Springer Verlag.

[2] Martín Abadi and Luca Cardelli. An imperative object calculus. Draft available on the WWW via the URL `http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers.html`., 1994.

[3] Martín Abadi and Luca Cardelli. A semantics of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science, Paris, France*, pages 332–341, Los Alamitos, CA, July 1994. IEEE.

[4] Martín Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, New York, NY, April 1994.

[5] Serge Abiteboul, Paris C. Kanellakis, and Emmanuel Waller. Method schemas (preliminary report). In *Principles of Data Base Systems, Nashville*, pages 16–27. ACM, 1990.

[6] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.

[7] Andrew Black and Jens Palsberg. Foundations of object-oriented languages: Workshop report. *ACM SIGPLAN Notices*, 29(3):3–11, March 1994. The bibliography was truncated in the published version. Obtain the full report by anonymous ftp from `crl.dec.com` in `pub/DEC/sigplan94.ps.Z`.

[8] K.B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[9] Kim B. Bruce, Angela Schuett, and Robert van Gent. A type-safe polymorphic object-oriented language. Obtain by anonymous ftp from `cs.williams.edu` in `pub/kim/PolyTOIL.dvi`., July 1994.

[10] G. Castagna. Covariance and contravariance: conflict without a cause. Technical Report liens-94-18, LIENS, October 1994. Available by anonymous ftp from `ftp.ens.fr` in file `/pub/dmi/users/castagna/covariance.dvi.Z`.

[11] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[12] Craig Chambers and Gary T. Leavens. Type-checking and modules for multi-methods. In *OOPSLA '94 Conference Proceedings, Portland, Oregon.*, volume 29 of *ACM SIGPLAN Notices*, pages 1–15. ACM, October 1994.

[13] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, NY, 1991.

[14] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. Application of oop type theory: State, decidability, integration. In *OOPSLA '94 Conference Proceedings, Portland, Oregon*, volume 29 of *ACM SIGPLAN Notices*, pages 16–30. ACM, October 1994.

[15] Martin Hofmann and Benjamin Pierce. Positive subtyping. Technical Report ECS-LFCS-94-303, Department of Computer Science, Uni-

Name	Organization	E-mail address
Abadi, Martín	Digital, SRC	ma@src.dec.com
Abiteboul, Serge	INRIA	Serge.Abiteboul@inria.fr
Bellucci, Roberto	LIENS	bellucci@dmi.ens.fr
Benzaken, Véronique	Université de Paris I - Sorbonne	Veronique.Benzaken@lri.fr
Bouladoux, Francois	ENS	Francois.Bouladoux@dmi.ens.fr
Bruce, Kim	Williams College	kim@cs.williams.edu
Cardelli, Luca	Digital	luca@src.dec.com
Castagna, Giuseppe	LIENS	castagna@dmi.ens.fr
Davies, Rowan	INRIA & CMU	rowan@cs.cmu.edu
Delobel, Claude	LRI	claude@o2tech.o2tech.fr
Dhara, Kishore	Iowa State University	dhara@cs.iastate.edu
Eifrig, Jonathan	The Johns Hopkins University	eifrig@cs.jhu.edu
Ghelli, Giorgio	University of Pisa	ghelli@di.unipi.it
Gunter, Carl	University of Pennsylvania	gunter@cis.upenn.edu
Leavens, Gary	Iowa State University	leavens@cs.iastate.edu
Longo, Giuseppe	CNRS-LIENS	longo@dmi.ens.fr
Milsted, Kathleen	France Telecom CNET	milsted@issy.cnet.fr
Moggi, Eugenio	DISI, Univ. di Genova	moggi@disi.unige.it
Palsberg, Jens	Aarhus University	palsberg@daimi.aau.dk
Pierce, Benjamin	LFCS, Univ. of Edinburgh	bcp@dcs.ed.ac.uk
Rémy, Didier	INRIA-Rocquencourt	Didier.Remy@inria.fr
Smith, Scott	The Johns Hopkins University	scott@cs.jhu.edu
Trifonov, Valery	The Johns Hopkins University	trifonov@cs.jhu.edu
Waller, Emmanuel	U. Paris Sud	Emmanuel.Waller@lri.fr

Table 1: Participants in the workshop.

versity of Edinburgh, Edinburgh, U.K., September 1994. An extended abstract will appear in the POPL'95 proceedings. Available by anonymous ftp from `ftp.dcs.ed.ac.uk` in file `pub/bcp/pos.ps.Z`.

[16] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium of Principles of Programming Languages, Portland, Oregon*, pages 138–150. ACM, January 1994.

[17] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* ACM, January 1995. To appear. Available by anonymous ftp from `ftp.daimi.aau.dk` in file `pub/palsberg/papers/popl95.ps.Z`.

[18] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.

[19] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Get by anonymous ftp from `ftp.dcs.ed.ac.uk` in `pub/bcp/friendly.ps.Z`. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[20] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346, New York, NY, April 1994. Springer-Verlag.

[21] E. Waller. Schema updates and consistency. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases, second International Conference, Munich, Germany*, volume 566 of *Lecture Notes in Computer*

Science, pages 167–188. Springer-Verlag, New York, NY, December 1991.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR94-22
Submission Date: November 23, 1994