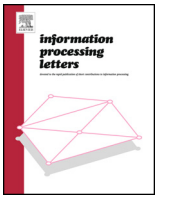
Information Processing Letters 178 (2022) 106285

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl



Static type checking without downcast operator

Arjan J. Mooij [a,b,*]

[a] ESI (TNO), Eindhoven, the Netherlands
[b] Delft University of Technology, Delft, the Netherlands

ARTICLE INFO

Article history:
Received 6 October 2020
Received in revised form 10 March 2021
Accepted 10 May 2022
Available online 16 May 2022
Communicated by Leah Epstein

Keywords:
Programming languages
Early validation
Object-oriented languages
Static type checking

ABSTRACT

In the last couple of years several dynamically-typed, object-oriented programming languages have been equipped with optional static type checkers. This typically requires these languages to be extended with a downcast operator, which is a common operator in statically-typed languages but not in dynamically-typed languages.
Our objective is to investigate an approach for static type checking of object-oriented languages that does not require such an additional downcast operator. We systematically weaken the rules for static type checking to avoid reporting errors that can be resolved using downcast operators. This leads to an approach similar to quasi-static typing that enables to make type annotations stricter in a gradual way.
These static type checking rules can be applied to dynamically-typed languages by interpreting the dynamic type as the top of the subtype relation. Based on these ideas we have implemented a static type checker for the dynamically-typed, object-oriented language POOSL without introducing a downcast operator. Practical experiences with this type checker indicate that it is useful for early validation.

1. Introduction

In the last couple of years widely-used, dynamically-typed programming languages such as Python and JavaScript have been equipped with optional static type checkers. Static type checking is known to be useful for machine-checkable documentation and early error detection [1–5]. Optional type systems [1] "*are neither syntactically nor semantically required, and have no effect on the dynamic semantics of the language*". The optional type checking required these object-oriented languages to be extended with a downcast operator, which can explicitly replace any static type (e.g., type *Object*) by a more specific subtype (e.g., type *String*). However, downcast operators are common in statically-typed languages but not in dynamically-typed languages.

The optional type hints of Python are described in PEP-484 [6], including the notion of type casts: "*Occasionally … the programmer may know that an expression is of a more constrained type than a type checker may be able to infer. For example:*

```
def find_first_str(a: List[object]) -> str:
    index = next(i for i, x in enumerate(a) if isinstance(x, str))
    return cast(str, a[index])
```

* Correspondence to: ESI (TNO), Eindhoven, the Netherlands.
E-mail address: Arjan.Mooij@tno.nl.

https://doi.org/10.1016/j.ipl.2022.106285

Some type checkers may not be able to infer that the type of a[index] is str …, but we know that … it must be a string. The cast(t, x) call tells the type checker that we are confident that the type of x is t. At runtime a cast always returns the expression unchanged – it does not check the type, and it does not convert or coerce the value."

The optional type annotations of TypeScript (that compiles to plain JavaScript) are described in [7], including the notion of type assertions: "*In a type assertion expression of the form* $< T > e$, *… the resulting type of e is required to be assignable to T, or T is required to be assignable to the widened form of the resulting type of e …. The type of the result is T. Type assertions check for assignment compatibility in both directions. Thus, type assertions allow type conversions that might be correct, but aren't known to be correct. … Type assertions are not checked at run-time.*"

Downcasts are used in the context of data structures without type generics (similar to the Python example), but also for language limitations such as missing support for polymorphism or for bridging the realms of static and dynamic typing. Other applications include the implementation of fragments of subtype-specific functionality, without adding a dedicated method to each subtype or introducing a dedicated visitor class.

Contribution. We systematically develop an approach for static type checking of object-oriented languages that does not require an additional downcast operator. Traditional approaches are based on a *subtype pre-order* that indicates that one type is a subtype of the other. In contrast, our approach is based on a symmetric *subtype compatibility* relation that indicates that two types have a common subtype. In the Python example, the explicit cast was used because type *object* is not a subtype of type *str*, but our approach avoids the cast as type *object* is compatible with type *str*.

The main contributions of this article are:

- Sect. 2: General approach for static type checking without downcast operator;
- Sect. 3: Design of a static type checker for the object-oriented language POOSL;
- Sect. 4: Evaluation of the usage of our optional static type checker for POOSL;
- Sect. 5: Extensive positioning of our general approach among related work.

Finally in Sect. 6 we draw some conclusions.

2. General approach: symmetric, non-transitive subtype compatibility

In this section we formalize static type checking, both with and without downcast operators. In particular we define the subtype compatibility relation, and study its formal properties. We also interpret these formal properties in terms of type checking.

2.1. Formalization

To model both nominal and structural subtyping in object-oriented languages, let relation $<:$ denote the *subtype pre-order* on types, such that $A <: B$ denotes that type A is a subtype of type B. (Recall that pre-orders are reflexive, transitive relations.) Variables, method input parameters and method return values can be *annotated* with a type B, to denote that the type of their dynamic value is expected to be a subtype of B.

Consider assigning a variable (or method return value) annotated with type A to a variable (or method input parameter) annotated with type B.

- Traditionally every subtype X of A is required to be a subtype of B:

  $$(\forall X : X <: A \Rightarrow X <: B)$$

  As $<:$ is a pre-order, this requirement is equivalent to the subtype pre-order:

  $$A <: B$$

  To make it easier to fulfill this requirement, downcast operators can be used to replace type A by any subtype of A. (Whether the downcast operator is checked dynamically is outside the scope of static type checking.)
- To avoid introducing downcast operators, our objective is to ignore violations that can be resolved using a downcast to a type X. This leads to a weaker requirement:

  $$(\exists X : X <: A \wedge X <: B)$$

  We use this as definition for the *subtype compatibility* relation $\sim$ on types:

  $$A \sim B$$

  Note that this derived definition of subtype compatibility is about the existence of any common subtype, but not necessarily a greatest common subtype.

(Downcast operators play no role in method declarations, and hence we do not need to consider the typical rule for overridden methods that the declared parameter and return value types should be equal, covariant or contravariant in the subtype pre-order.)

2.2. General properties

Subtype compatibility $\sim$ is reflexive and symmetric, which follows from its definition and the reflexivity of the subtype pre-order $<:$. The reflexivity is shared with the subtype pre-order, but the symmetry is an extra property. In contrast to the subtype pre-order, subtype compatibility is not guaranteed to be transitive.

As observed before [3,4], "*implicit downcasts combined with the transitivity of subtyping creates a fundamental problem that prevents the type system from catching all type errors*". In our approach the implicit downcasts do not affect the subtype pre-order $<:$ (which is transitive) but the subtype compatibility relation $\sim$ (which is not guaranteed to be transitive).

Some properties of subtype compatibility in relation to the subtype pre-order are:

- Subtype compatibility is an extension of the subtype pre-order: $A <: B \;\Rightarrow\; A \sim B$. This means that type checks that are valid according to traditional subtype rules are also valid according to subtype compatibility.
- Every top element of the subtype pre-order (i.e., every type that is a supertype of all types) is subtype compatible with each element. This means that top elements can be used to model the dynamic type when integrating static and dynamic typing.
- Each side of the symmetric subtype compatibility relation is monotonous with respect to the subtype pre-order: $A <: B \;\Rightarrow\; (A \sim C \;\Rightarrow\; B \sim C)$. This relates to the gradual guarantee [5]: "*if a gradually typed program is well typed, then removing type annotations always produces a program that is still well typed*".

Each subtype incompatibility that involves a type annotation (for a variable or method) can be resolved by replacing the annotation by a common supertype of the incompatible types. As this cannot introduce new incompatibilities, iterative replacements form a strategy to resolve all subtype incompatibilities that involve type annotations.

2.3. Additional properties for nominal subtyping with single inheritance

In the special case of nominal subtyping with single inheritance, for every type X, every two supertypes A and B of type X are related in terms of the subtype pre-order, i.e., $X <: A \;\wedge\; X <: B \;\Rightarrow\; A <: B \;\vee\; B <: A$. In such cases subtype compatibility is related to the subtype pre-order as follows:

$$A \sim B \;\equiv\; A <: B \;\vee\; B <: A$$

This means that subtype compatibility can be interpreted as sharing a branch in the directed tree corresponding to the subtype pre-order.

3. Design of a static type checker for POOSL

We have applied our general static type checking approach to the language POOSL (Parallel Object-Oriented Specification Language), which is a dynamically-typed, imperative language without a downcast operator. The syntax and semantics of this language have been formally defined and analyzed [8,9]. In this section we present our developed static typing rules and some of the features implemented in our optional type checker.

3.1. Static typing rules

The POOSL syntax requires type annotations in the declaration of variables and methods (viz., for their input parameters and return value), but these annotations have no formal semantics. Types include pre-defined types (e.g., *Object*, *Boolean*, and *Integer*) and user-defined types with single inheritance. For the subtype pre-order we use the nominal inheritance relation with type *Object* as the unique top element.

For every expression E we compute a single type T given a context Γ, which is denoted by typing judgment $\Gamma \vdash E : T$. The context consists of type annotations of variables (e.g., $v : T$) and methods (e.g., $m : \vec{S} \to T$, where $\vec{S}$ denotes the list of types for the receiver object and input parameters). We use the binary operator $\uparrow$ on pairs of types to denote the strictest common supertype in the subtype pre-order, and similarly we use the unary operator $\uparrow$ on non-empty sets of types. We use the binary relation $\sim$ on types to denote subtype compatibility as defined in Sect. 2.1; as a shorthand, $\vec{R} \sim \vec{S}$ denotes that the lists $\vec{R}$ and $\vec{S}$ have the same length and are point-wise subtype compatible.

Fig. 1 shows our static type inference rules for some typical POOSL expressions. The set of rules is not complete, but illustrates the approach developed in Sect. 2.1. Value **nil** represents the undefined object, which could have every type; as top type *Object* is compatible with every type, we use type *Object* for value **nil**. Rule MethodCall uses a list $\vec{R}$ for the types of the values in the call (the receiver $\vec{E}_0$ and the arguments $\vec{E}_1$ ... $\vec{E}_n$), and then considers all method declarations with

$$\text{BooleanLiteral}\ \frac{b \text{ is a boolean literal}}{\Gamma \vdash b : \mathit{Boolean}} \qquad \text{IntegerLiteral}\ \frac{i \text{ is an integer literal}}{\Gamma \vdash i : \mathit{Integer}}$$

$$\text{NilLiteral}\ \frac{}{\Gamma \vdash \textbf{nil} : \mathit{Object}} \qquad \text{New}\ \frac{T \text{ is a non-primitive type}}{\Gamma \vdash \textbf{new}(T) : T}$$

$$\text{Variable}\ \frac{(v : T) \in \Gamma}{\Gamma \vdash v : T} \qquad \text{Assignment}\ \frac{(\Gamma \vdash v : S) \quad (\Gamma \vdash E : T) \quad S \sim T}{\Gamma \vdash v := E : T}$$

$$\text{IfThenElse}\ \frac{(\Gamma \vdash C : R) \quad R \sim \mathit{Boolean} \quad (\Gamma \vdash E : S) \quad (\Gamma \vdash F : T)}{\Gamma \vdash \textbf{if } C \textbf{ then } E \textbf{ else } F \textbf{ fi} : S \uparrow T}$$

$$\text{MethodCall}\ \frac{(\forall i : (\Gamma \vdash \vec{E}_i : \vec{R}_i)) \quad (\exists \vec{S}, T : \vec{R} \sim \vec{S} \ \wedge\ (m : \vec{S} \to T) \in \Gamma)}{\Gamma \vdash \vec{E}_0\ m(\vec{E}_1, ..., \vec{E}_n) : \uparrow \{T \mid \vec{S}, T : \vec{R} \sim \vec{S} \ \wedge\ (m : \vec{S} \to T) \in \Gamma\}}$$

Fig. 1. Example Static Typing Rules for POOSL based on Nominal Typing.

Assignments:

```
o := s;  // Implicit up-cast
i := o;  // Implicit down-cast
i := s;  // Incompatible
```

Method Calls and Gradual Refinement:

```
elements clear();
elements append(42);
i := elements at(1);
```

Fig. 2. Elementary Examples.

name m and compatible types $\vec{S}$ to compute the strictest common supertype of their return types T. Note that there is no subsumption rule to replace the type of a typing judgment by a supertype.

Afterthought. Suppose that we use a subtype pre-order based on nominal subtyping with multiple inheritance (which is not supported in POOSL), or based on structural subtyping (which is closer to the dynamic semantics of POOSL). Then there may not be a unique top element in the subtype pre-order, and there may not be a unique strictest common supertype operator. We expect that these issues can be circumvented by typing judgments that compute a set of types. Then the strictest common supertype on sets of types could be defined as $\mathbb{S} \uparrow \mathbb{T} = \mathbb{S} \cup \mathbb{T}$, and subtype compatibility on sets of types as $\mathbb{S} \sim \mathbb{T} = (\exists S, T : S \in \mathbb{S} \ \wedge\ T \in \mathbb{T} \ \wedge\ S \sim T)$. We do not explore this any further.

3.2. Warnings and quick-fixes

We have implemented the static typing rules from Fig. 1 in the POOSL IDE (see https://github.com/eclipse/poosl). This static type checker gives warnings instead of errors, because the type annotations have no formal dynamic semantics in POOSL.

Regarding the type compatibility premises, only the one from rule IfThenElse may not involve a type annotation (from a variable or method). To solve type incompatibilities from the other rules, the strategy from Sect. 2.2 can be applied. For warnings from the Assignment rule we provide a quick-fix that replaces the type S of variable v by $S \uparrow T$.

4. Usage of the static type checker for POOSL

In this section we provide some elementary examples for our static type checking approach, and briefly discuss our practical experience with real POOSL models.

4.1. Elementary examples

The examples in Fig. 2 use variables o, i and s which are declared to be of type *Object*, *Integer* and *String* respectively. All types are a subtype of type *Object*, but types *Integer* and *String* are unrelated in the subtype pre-order. The types *Object* and *Integer* are subtype compatible, and the types *Object* and *String* are subtype compatible. However, the types *Integer* and *String* are not subtype compatible. This illustrates that subtype compatibility is symmetric but not transitive.

At the left-hand side of Fig. 2, traditionally only implicit up-casts (e.g., $o := s$) are allowed. Using our approach both implicit up-casts (e.g., $o := s$) and implicit down-casts (e.g., $i := o$) are allowed. However, mixing incompatible types in an assignment (e.g., $i := s$) is still not allowed; the associated quick-fix would replace the type of variable i by the strictest common supertype of types *Integer* and *String*, which is type *Object*.

The right-hand side of Fig. 2 uses an object-oriented hierarchy (without type generics) of data structures: the types *Sequence* and *Set* are subtypes of the type *Collection*, which in turn is a subtype of the type *Object*. The methods *append* and *at* are only defined on receivers of type *Sequence*, and hence traditionally the variable *elements* must be declared to be of the type *Sequence*. Using our approach, the calls to these methods result in type violations if the variable *elements* is statically declared to be of the type *Set*, but not if it is declared to be of the type *Sequence*, *Collection*, or *Object*.

This also shows that static type annotations can be introduced gradually (e.g., in the context of dynamic typing): initially all variables can have the type *Object*, then refine the variable *elements* to the type *Collection*, and finally refine the variable *elements* to the type *Sequence*. In contrast to traditional static type checking, after every refinement step there are no type violations using our static type checking approach.

4.2. Practical experience

Our work on an optional static type checker for POOSL started after observing many errors like "method *x* does not exist on object of type *y* with arguments of type *z*" during the simulation of our industrial POOSL models [10]. Apart from repairing the model, such an error also requires restarting the simulation environment (which included external tools) and running again the analysis scenario of interest.

These models primarily use elementary types like *Boolean*, *Integer* and *String*, which are all direct subtypes of the type *Object*. In particular the type *Integer* has a method + for addition (with an *Integer* argument), whereas the type *String* has a method + for concatenation (with a *String* argument). Conversions between the type *Integer* and the type *String* require explicit method calls that can easily be forgotten in practice.

Since the first implementation [11] of our static type checking ideas, we experience that such errors during simulation have practically disappeared. In addition we observe no false type errors when modeling and using an object-oriented hierarchy (without type generics) of data structures like queue, stack, map, bag, set, and sequence.

On the other hand, as explored in Sect. 2.2, we realize that our static type checker for POOSL does not give many formal guarantees. This relates to experiences reported in the context of TypeScript [12]: "*an unsound type system can still be extremely useful*" and "*while the type system can be wrong about the shape of run-time structures, the experience thus far indicates that it usually won't be*".

5. Related work for our general approach

The goal of related work like [13] is to (symmetrically) integrate static and dynamic typing, but usually they focus on introducing dynamic typing in a statically-typed language. In contrast we focus on languages without downcast operators, which are often dynamically-typed languages, in which static type-checking is introduced. In what follows we discuss four related approaches in more detail.

5.1. Type annotation

In this subsection we describe two approaches that rely on given type annotations, which is similar to our setting.

5.1.1. Gradual typing: implicit casts with auxiliary "unknown" type

Gradual typing [3–5] intends to enable programmers to gradually introduce static type checking in a dynamically-typed language. Their approach is to introduce an auxiliary *unknown* type denoted by "?" that is neutral to subtyping (i.e., only $? <: ?$), and to introduce a *consistent-subtyping* relation on types which only applies the subtyping relation on the structural parts that are known in both types. Thus "*type consistency and subtyping are orthogonal and can naturally be superimposed*". Most papers [3,4] on gradual typing focus on structural type systems, but there exists earlier work [14] with similar ideas for nominal type systems.

Program fragments that do not contain type *unknown* are checked at compile-time, and program fragments that do contain type *unknown* are checked at run-time. Implicit casts from unknown to known types may fail at run-time (like a downcast), but these implicit casts form the distinguishing feature of gradual typing.

In comparison to gradual typing, our approach does not have the orthogonality property, but it does allow a more gradual/iterative transition from the *unknown* type to various subtypes by encoding the *unknown* type as the top of the subtype pre-order. Implicit casts, in our case traditional downcasts, are also at the heart of our approach. On the other hand, we leave it open whether the explicit type annotations and implicit casts have an effect on the dynamic semantics (that is, that they are actually checked at run-time), or that they are just a form of machine-checked annotations.

Abstracting gradual typing [15] is a generalization that covers also our approach, by seeing our gradual types as bounded types. Potential extensions of our gradual types include static types for literals and if-conditions, and option types for values like **nil**.

5.1.2. Quasi-static typing: implicit casts with top "unknown" type

Quasi-static typing [16] uses a top type Ω to represent the unknown type (i.e., $t <: \Omega$ for all types t) and they propose to use implicit downcasts. The type checking consists of a two-phase approach. In the first phase, a minimal combination of required type casts is computed; for example, a function application may require a downcast of the parameter value to a less-dynamic expected parameter type. All programs are accepted in the first phase, but normally ill-typed programs are accepted with required downcasts. The second phase consists of plausibility checking of the computed minimal combination of upcasts and downcasts. Their focus is on structural subtyping. In terms of number of citations, quasi-static typing seems to be less popular than gradual typing.

In comparison to quasi-static typing, our approach looks semantically very similar, but technically much simpler. The simplicity aspect is very relevant, as concerns have been raised [3] about the technical details of quasi-static typing.

5.2. Type inference

In this subsection we describe two approaches that rely on type inference. This differs from our approach as we consider the type annotations as useful (machine-checkable) documentation that is checked for internal consistency.

5.2.1. Soft typing: report all possible errors

Soft typing [17] is a form of static type checking that is focused on reporting possible type errors. The key idea is to never reject any program, but to insert explicit run-time checks for possible type errors. Following a minimal-failure principle, the aim is to minimize the number of inserted run-time checks. Following a minimal-text principle, sophisticated type inference algorithms are used such that the programmer does not need to declare types in the program.

In comparison to soft typing, our approach does not aim to report all possible errors as we intentionally ignore downcast-related errors. Moreover we leave it open whether the type annotations have an effect on the dynamic semantics. Our optional static type checker for POOSL only produces warnings, and hence also never rejects any program. In the spirit of their minimal-text principle, our approach does not require the type annotations to be strict (for example, all type annotations can use the top type).

5.2.2. Success typing: report only certain errors

Success typing [2] is related to soft typing [17] as both approaches never reject any program. However, they are opposites with respect to the reported type errors. To eliminate false reports, success typing only reports certain type errors. This is particularly visible in their App rule, that states that the result of function application is a subtype of the declared return type of the function.

In comparison to success typing, our approach also avoids uncertain type errors, in particular related to missing downcast operators. The App rule from success typing seems closely related to the issue of missing downcast operators.

6. Conclusions

We have presented a general approach for the static type checking of object-oriented languages without requiring an additional downcast operator. We have constructed this approach by systematically weakening the rules for static type checking to avoid reporting errors that can be resolved using downcast operators (if they were available). This leads to a simpler characterization of an approach similar to quasi-static typing. We have shown that the approach enables novel strategies for solving type incompatibilities, and enables to make type annotations stricter in a gradual way.

Our study takes the perspective of absent downcast operators instead of modeling the dynamic type. The frequently studied conversions from the dynamic type to static types can be seen as a downcast by interpreting the dynamic type as the top of the subtype pre-order. Based on these ideas we have implemented in the POOSL IDE a static type checker for the dynamically-typed, object-oriented language POOSL. Our practical experiences with this type checker indicate that it is useful for early validation.

CRediT authorship contribution statement

Arjan Mooij: Conceptualization, Methodology, Software, Writing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The author likes to thank Jozef Hooman for sharing his experiences with our static type checker for POOSL. He also likes to thank Eelco Visser, Jeff Smits and the anonymous referees for their feedback on earlier versions of this work.

References

[1] G. Bracha, Pluggable type systems, in: Proceedings of the OOPSLA'04 Workshop on Revival of Dynamic Languages, ACM Press, 2004.
[2] T. Lindahl, K. Sagonas, Practical type inference based on success typings, in: Proceedings of PPDP'06, ACM, 2006, pp. 167–178.
[3] J.G. Siek, W. Taha, Gradual typing for functional languages, in: Proceedings of Scheme'06, University of Chicago, 2006, pp. 81–92. Technical Report TR-2006-06.
[4] J.G. Siek, W. Taha, Gradual typing for objects, in: Proceedings of ECOOP'07, Springer, 2007, pp. 2–27.

[5] J.G. Siek, M.M. Vitousek, M. Cimini, J.T. Boyland, Refined criteria for gradual typing, in: Proceedings of SNAPL'15, in: LIPIcs, vol. 32, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 274–293.
[6] G. Van Rossum, J. Lehtosalo, L. Langa, Type Hints, Python Enhancement Proposal (PEP), vol. 484, 2014, https://www.python.org/dev/peps/pep-0484/.
[7] Microsoft, TypeScript Language Specification, 1.8 edition, 2016, https://github.com/microsoft/TypeScript/tree/master/doc/.
[8] P.H.A. van der Putten, J.P.M. Voeten, Specification of Reactive Hardware/Software System, Ph.D. thesis, Eindhoven University of Technology, 1997.
[9] L. van Bokhoven, Constructive Tool Design for Formal Languages; From Semantics to Executing Models, Ph.D. thesis, Eindhoven University of Technology, 2004.
[10] J. Hooman, A.J. Mooij, H. van Wezep, Early fault detection in industry using models at various abstraction levels, in: Proceedings of IFM 2012, in: LNCS, vol. 7321, Springer, 2012, pp. 268–282.
[11] Z. Manevska, User Interaction Layer for POOSL, Technical Report, Stan Ackermans Institute, 2012, Supervisors: J. Hooman, A.J. Mooij, R. Kuiper.
[12] G.M. Bierman, M. Abadi, M. Torgersen, Understanding TypeScript, in: Proceedings of ECOOP 2014, in: LNCS, vol. 8586, Springer, 2014, pp. 257–281.
[13] M. Abadi, L. Cardelli, B.C. Pierce, G.D. Plotkin, Dynamic typing in a statically-typed language, in: Proceedings of POPL'89, ACM Press, 1989, pp. 213–227.
[14] C. Anderson, S. Drossopoulou, BabyJ: from object based to class based programming via types, in: Proceedings of WOOD'03, in: ENTCS, vol. 82, Elsevier, 2003, pp. 53–81.
[15] R. Garcia, A.M. Clark, É. Tanter, Abstracting gradual typing, in: Proceedings of POPL'16, ACM, 2016, pp. 429–442.
[16] S.R. Thatte, Quasi-static typing, in: Proceedings of POPL'90, ACM Press, 1990, pp. 367–381.
[17] R. Cartwright, M. Fagan, Soft typing, in: Proceedings of PLDI'91, ACM, 1991, pp. 278–292.