Gradual Typing for Annotated Type Systems

Peter Thiemann and Luminous Fennell

University of Freiburg, Germany
{fennell,thiemann}@informatik.uni-freiburg.de

Abstract. Annotated type systems include additional information in types to make them more expressive and to gather intensional information about programs. Gradual types enable a seamless transition between statically and dynamically checked properties of values. Gradual annotation typing applies the ideas of gradual typing to the annotation part of a type system.

We present a generic approach to transform a type system with annotated base types into a system which gradualizes the information contained in the annotations. We prove generic type safety for the gradualized extensions and similar generic versions of the blame theorem for a calculus with run-time annotations. We relate this calculus to a more efficient calculus which elides run-time annotations in the statically annotated parts. We further introduce and prove correct a syntactic transformation that eliminates run-time annotation tests by enlarging the statically annotated parts.

1 Introduction

Refinement type systems have been proposed by a number of researchers to sharpen the guarantees of existing type systems. Examples are Freeman and Pfennings's system to distinguish empty and non-empty lists by type [8], Pessaux and Leroy's exception analysis [21], Jackson's dependency analysis [14], Chin and coworkers' type qualification framework [3], and many others.

In each case, the type language is extended with annotations that either abstract semantic properties of values beyond the capabilities of the underlying type language or they express properties that are not locally checkable. An example of the latter kind are type systems for dimension analysis [15,23] that formalize a notion of parametricity that cannot be checked on single values [16]. Haskell DSLs that employ phantom types provide further examples [18].

Annotated type and effect systems [20,29] play an important role in program analysis where the annotations serve to express intensional information about data. Example uses are race detection and locking [7,1] and, prominently, information flow analysis [10] (using just one example of many). Languages like Java include annotation frameworks that are often used dynamically. An instance of such a framework could be promoted to gradual checking using a suitable extension of our approach. Similar ideas have been pursued in the past [30,4].

Gradual typing [27,25] is concerned with controlling the boundary between static and dynamic typing. A gradual type system provides cast operations that

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 47–66, 2014.

reveal sufficient static type information to improve the efficiency of otherwise dynamically typed programs.

Recent work has considered a number of variations on gradual typing that are not directly related to dynamic typing. For example, Disney and Flanagan [5] as well as Fennell and Thiemann [6] consider gradual information flow, Wolff and coworkers consider gradual typestate [33], and Sergey and Clarke proposed gradual ownership types [24], which are further discussed in the related work.

This proliferation of gradual systems begs the question if there is a common structure underlying all these gradual systems. In this work, we give a partial answer by outlining **a generic approach to "gradualize" existing annotated type systems that support annotations on base types.** Our proposed method is geared towards functional programming, therefore it cannot be expected to handle the gradualized object-oriented systems [33,24] (for example, gradual typestate requires dealing with linearity, which we did not consider).

Scope and limitations: Our approach applies to all properties that can be expressed by additional tokens on base-type values at run time: dimensions, phantom types, security labels, sanitization, representation restrictions (e.g., serializable), and so on. Extensional properties (e.g., refinements) that may be expressed with a predicate as in a subset type $\{x \in B \mid \Phi(x)\}$ are also expressible in our framework by taking a set of predicates as annotations. However, run-time tokens are not needed for establishing a gradual system with subset types because the predicates may just be checked during run time. In exceptional cases, checking a predicate at run time may be too expensive, in which case our approach could be used to handle a run-time token that asserts Φ.

Contributions. We claim that the essence of gradualization for an annotated type system consists of (a) specifying a calculus with run-time annotations and checking, (b) introducing cast operations to stage annotation checking, and (c) eliminating the statically checked annotations. We substantiate this claim in the context of a statically-typed call-by-value lambda calculus, where only base types carry annotations. For this calculus, we prove type soundness and a blame theorem (roughly: only casts at the dynamic→static boundary can fail).

We discuss two approaches to simplify run-time annotations. One of them yields an efficient run-time model where statically checked annotations are erased.

We propose a compile-time transformation to eliminate run-time checks and prove its correctness.

Overview. After some motivating examples (Section 2), we specify a generic base-type annotated type systems and prove generic type safety and blame theorems (Section 3). Subsections 3.5 and 3.6 discuss alternative treatments of annotations including erasure. Section 4 introduces and proves correct the transformation rules to decrease the amount of dynamic checking. We wrap up with a discussion of related work (Section 5) and a conclusion.

2 Gradual Refinement Typing at Work

We demonstrate how gradual typing can remedy problems with overly conservative type-checking in two scenarios: a type system with dimension analysis and a type system that distinguishes encrypted and plaintext data.

2.1 Dimensions

Type systems with dimensions guard the programmer from mixing up measurements of different dimensions that are represented with a common numeric type [15]. For illustration we consider an ML-like language with simple types where numbers carry a dimension annotation. The following function, calculating an estimated time to arrival, is well-typed in this language.

```
fun eta (dist : float[m]) (vel:float[m/s]) : float[s] =
  dist / vel
```

The annotated type `float[u]` represents an integer of dimension `u` where `u` ranges over the free abelian group generated by the SI base dimensions: `m`, `s`, `kg`, and so on. The neutral element is written as `1`. The next example does not type check, because the typing of `_ - _` requires the same dimension for both arguments.

```
fun eta_broken (dist : float[m]) (vel : float[m/s]) =
  dist - vel
```

Each gain in safety costs flexibility. Thus, all published dimension type systems support dimension polymorphism. However, there are examples where polymorphism is not sufficient as in the definition of the power function on meters.[1]

```
fun pow_m (x : float[m]) (y : int[1]) =
    if y == 0 then 1[S(1)] else x * pow_m x (y - 1)
```

This definition does not type-check in a system based on simple types. Polymorphism does not help, either, because the dimension of the result depends on the parameter y as in `float[m`^{y}`]`. Nevertheless, `pow_m` is useful to define generic operations on geometric objects, like the n-dimensional volume and the $n-1$-dimensional surface of an n-dimensional hypercube given its base length `c`:

```
fun  nVolume (n : int[1]) (c : float[m]) =
  pow_m c n
fun  nSurface (n : int[1]) (c : float[m]) =
  (2 * n) * nVolume (n-1) c
```

A gradual annotation for such functions avoids the complexity of dependent types and preserves some guarantees about the annotation. In our system, the function `pow_m` could be modified to have type

```
pow_mg : float[m] → int[1] → float[?]
```

[1] The annotation `S(1)` indicates a statically checked dimensionless number.

The `?` annotation marks the annotation of the result type as *dynamic* and indicates that the run-time system needs to check the consistent use of the dynamic dimension of the value. The programmer has to insert casts of the form $e : t \rightsquigarrow t'$, where t is the type of e and t' is the destination type. Casts only switch type annotations from static to dynamic or vice versa. Here is the implementation of `pow_m` in the gradual system:

```
1 fun pow_mg (x : float[m]) (y : int[1]) =
2     if y == 0 then 1[D(1)]
3     else (x : float[m] ⇝ float[?]) * pow_mg x (y - 1)
```

The cast `x : float[m]` $\rightsquigarrow$ `float[?]` in line 3 converts `x` of type `float[m]` to destination type `float[?]` with a dynamic dimension initialized by the dimensionless `1[D(1)]`. At run time, values of dynamic dimension are marked with a `D`, as illustrated in line 2. The dynamically annotated result can be reintegrated into statically verified code by casting the dynamic annotation to a static one:

```
fun volume3d : float[m³] =
  (nVolume 3 2[m]) : float[?] ⇝ float[m³]
```

While it is possible to write type incorrect programs that cannot be sensibly executed, the run-time system rejects illegal casts. For example, the expression `(nVolume 3 2[m]) : float[?]` $\rightsquigarrow$ `float[m²]` evaluates to `8[D(m³)] : float[?]` $\rightsquigarrow$ `float[m²]`. As the computed dimension `D(m³)` is incompatible to the expected dimension `m²`, the cast fails and stops a computation with a potentially flawed result.

2.2 Tracking Encrypted Data

Custom type annotations are also useful to track certain operations on data throughout the program. As an example, consider the following program fragment that operates on encrypted as well as plaintext data.

```
1  val prog (encrypt : int → int)
2           (decrypt : int → int)
3           (inc : int → int)
4           (sendPublic : int → unit)
5           (displayLocal : int → unit)
6           (v : int) : unit =
7      displayLocal (decrypt v)
8      let v' = inc (decrypt v) in
9      sendPublic (encrypt v')
10     let v'' = ... in
11     sendPublic v''
```

It is parameterized by the operations for encryption, decryption, and increment and also receives a value. It is crucial that the operations are not applied arbitrarily: only encrypted data should be sent over the public channel (lines 9 and 11), incrementation only yields a sensible result on plaintext data (line 8), and only encrypted values should be decrypted to avoid gibberish (line 7).

If such a program grows sufficiently complex, these restrictions should be checked in a principled way. A lightweight way of doing so is to add suitable annotations to the type language and have them statically checked as much as possible. The types in the signature of `prog` could be enhanced with annotations indicating whether a number is encrypted ($\bullet$) or in plaintext ($\circ$).

```
val prog (encrypt : int° → int•)
         (decrypt : int• → int°)
         (inc : int° → int°)
         (sendPublic : int• → unit)
         (displayLocal : int° → unit)
         (v : int•) : unit = ...
```

A programmer can easily program against such an annotated signature. However, there might be legacy code that might not fit the more restrictive annotated typing discipline, even if it performs correctly dynamically. For example, the following procedure uses a boolean flag in order to distinguish encrypted data from plaintext:

```
fun prog' (is_encrypted : bool)
          encrypt decrypt inc sendPublic displayLocal v =
  let v' = if is_encrypted
           then encrypt (inc (decrypt v))
           else inc v
  if is_encrypted then sendPublic v' else displayLocal v'
```

Most type systems ignore conditional control flow and therefore would reject `prog'`. But, as in Section 2.1, it is possible to use the gradual typing approach for programs that are written in such a "dynamic style" by inserting suitable casts:

```
val prog_safe : (int° → int•) → (int• → int°) →
                (int° → int°) →
                (int• → unit) → (int° → unit) →
                int• → unit =
  (prog' true) : ((int? → int?) → (int? → int?) → ...)
                  ⇝ ((int° → int•) → (int• → int°) → ...)
```

The last line casts the legacy program `prog'` to the type of `prog_safe`. All interface functions passed to `prog'` are assumed to accept and return dynamic numbers of type $\texttt{int}^?$. To work correctly, the program `prog'` has to be recompiled with the gradual type on the left. The gradual annotated type system accepts `prog_safe` and the run-time system checks the correct use of the encryption operations dynamically. The underlying unannotated type system still rules out type errors on arithmetic operations, like calling `prog'` with a string as last argument.

3 The Generic Calculus with Base Type Annotations

The generic calculus λ^{BA} relies on a *base type annotation algebra* $\mathcal{A}$ with the same signature $\Sigma = (\oplus, \dots)$ as the primitive operations on base types.

$$t ::= B[a] \mid t \to t \qquad v ::= b[a] \mid \lambda x.e$$
$$e ::= b[a] \mid e \oplus e \mid x \mid \lambda x.e \mid e\,e \qquad E ::= [\,] \mid E \oplus e \mid v \oplus E \mid E\,e \mid v\,E$$

Fig. 1. Syntax: types, expressions, values, evaluation contexts

Thus, $\mathcal{A} = (A, \oplus_A, \dots)$ where A is the carrier set and each $(\oplus_A) : A \times A \hookrightarrow A$ is a partial function on A. Partiality is needed, e.g., for dimension analysis where addition is only sensible for arguments with the same dimension.

3.1 Static Annotated Typing

Figure 1 defines the syntax of λ^{BA}. A type t, is either a base type B annotated with an annotation $a \in A$, a function type, or any other standard type. In the term language e, base type values b carry a corresponding annotation. The remaining term constructors are as usual. Values v and evaluation contexts E are defined in the standard way.

Lambda expressions are interpreted as call-by-value functions, hence they reduce with the β_v reduction rule where $e[x \mapsto v]$ denotes the capture-avoiding substitution of v for x in e.

$$\text{BA-S-BetaV}\quad (\lambda x.e)\,v \longrightarrow e[x \mapsto v]$$

The evaluation of primitive operations is governed by another Σ algebra $(B, \oplus_B, \dots)$ where, again, $(\oplus_B) : B \times B \hookrightarrow B$ is a partial function. The dynamics for $\oplus$ check if the annotations of the arguments are combinable with $\oplus_A$ and execute the operation using its interpretation $\oplus_B$ on base-type values. We write $b_1 \oplus_B b_2 =: b$ as a shorthand for $(b_1, b_2) \in \mathit{dom}(\oplus_B)$ and $b_1 \oplus_B b_2 = b$.

$$\text{BA-S-Op}\quad \frac{b_1 \oplus_B b_2 =: b \qquad a_1 \oplus_A a_2 =: a}{b_1[a_1] \oplus b_2[a_2] \longrightarrow b[a]}$$

This rule may fail for two reasons, either the annotations are incompatible $(a_1, a_2) \notin \mathit{dom}(\oplus_A)$ or the operation is not defined on the particular argument values, i.e., $(b_1, b_2) \notin \mathit{dom}(\oplus_B)$. The example of dimension analysis demonstrates that the two conditions are independent. In the computation $3[m]/0[m]$, the division of the dimensions is defined, but $3/0$ is undefined.

The corresponding typing rule checks the annotations and the rule for constants just matches the annotations.

$$\text{BA-T-Const}\quad \Gamma \vdash b[a] : B[a] \qquad \text{BA-T-Op}\quad \frac{\Gamma \vdash e_1 : B[a_1] \qquad \Gamma \vdash e_2 : B[a_2] \qquad a_1 \oplus_A a_2 =: a}{\Gamma \vdash e_1 \oplus e_2 : B[a]}$$

Type soundness of the annotated type system implies that well-typed operations make the run-time check on annotations in rule BA-S-Op obsolete. Consequently the run-time annotations on well-typed programs could be erased. The erasure of statically verified annotations is further discussed in Section 3.6.

3.2 Gradual Annotated Typing

Our execution model from Subsection 3.1 equips all base-type values with run-time value annotations. For gradualization, we transition to a calculus λ_G^{BA} where value annotations are categorized as either static or dynamic and the operations on them are lifted from the original annotation algebra. Subsections 3.5 and 3.6 discuss the drawbacks of alternative approaches and demonstrate how the efficiency of annotation handling at run time can be improved.

Before we continue, it is important to realize that gradual annotation typing is different to gradual typing or dynamic typing. In dynamic typing, primitive operations, like addition, have a fixed low-level type, say, `int->int->int`. To execute these operations requires dynamic arguments to be unwrapped and results to be wrapped in a dynamic container. For that reason, gradual typing [27] starts with a type system that exposes these low-level types and introduces casts to revert to type dynamic if the low-level types do not match.

In annotated gradual typing, we take the low-level typing of operations for granted: an addition on integers may certainly be executed, **but** it may be forbidden because of non-matching dimension annotations, say. In particular, it is not desirable to even define a translation that introduces casts because the same addition operation may be used polymorphically with arguments of different (but matching) dimensionality.

Gradualization requires two different extensions of the annotation algebra, one for type annotations and one for value annotations. Type annotations in the gradual system, *ta*, are drawn from $\mathcal{A}^? = (A^?, \oplus, \dots)$ where $A^? = A \cup \{?\}$ and an operation is lifted from $\mathcal{A}$ by insisting that any ? argument makes the result ?, or that all arguments are in A, in which case the operation works as before.

$$
\begin{aligned}
? \ \oplus_{A^?} \_ \ &= ? \\
\_ \ \oplus_{A^?} ? \ &= ? \\
a_1 \oplus_{A^?} a_2 &= a \quad a_1, a_2 \in A, (a_1 \oplus_A a_2) =: a
\end{aligned}
$$

Apart from drawing type annotations from this extended algebra, the type language is unchanged.

The refined algebra $\mathcal{A}^+ = (A^+, \oplus, \dots)$ stages the value annotations using $A^+ = \mathbf{D}(A) + \mathbf{S}(A)$, the disjoint union of two copies of A tagged with $\mathbf{D}$ and $\mathbf{S}$, where $\mathbf{D}$ annotations are only checked dynamically and $\mathbf{S}$ annotations are (also) checked statically. The operations are lifted to $\mathcal{A}^+$ by insisting that results are static unless any dynamic argument is present. In any case, they apply the underlying operation from $\mathcal{A}$.

$$
\begin{aligned}
\mathbf{D}(a_1) \oplus_{A^+} V(a_2) &= \mathbf{D}(a) \quad (a_1 \oplus_A a_2) =: a \\
V(a_1) \oplus_{A^+} \mathbf{D}(a_2) &= \mathbf{D}(a) \quad (a_1 \oplus_A a_2) =: a \\
\mathbf{S}(a_1) \oplus_{A^+} \mathbf{S}(a_2) &= \mathbf{S}(a) \quad (a_1 \oplus_A a_2) =: a
\end{aligned}
$$

Here and in the following, the meta variables $V, V_1, V_2, \dots$ range over $\mathbf{D}$ and $\mathbf{S}$ and meta variables $va, va_1, \dots$ range over annotations of the shape $V(a)$.

The term language is extended by type (annotation) casts.

$$e ::= \cdots \mid e : t \rightsquigarrow^p t$$

$$\frac{b_1 \oplus b_2 =: b \qquad va_1 \oplus_{A^+} va_2 =: va}{b_1[va_1] \oplus b_2[va_2] \longrightarrow b[va]}\ \text{BA-SG-Op}$$

$$\frac{V_1(a) \prec ta_1 \qquad V_2(a) \prec ta_2}{(b[V_1(a)] : B[ta_1] \rightsquigarrow^p B[ta_2]) \longrightarrow b[V_2(a)]}\ \text{BA-SG-Cast-Base}$$

$$v : (t_1 \to t_2) \rightsquigarrow^p (t_1' \to t_2') \longrightarrow \lambda x.(v(x : t_1' \rightsquigarrow^{\overline{p}} t_1)) : t_2 \rightsquigarrow^p t_2'\ \text{BA-SG-Cast-Fun}$$

Fig. 2. Dynamics of the gradual annotation calculus

They modify the annotations but leave the shape of types intact. The *blame label* p on the cast indicates the source of the potential error. Blame labels come with an involutory operation $\overline{\cdot}$ that flips the polarity of the blame between positive $p = \overline{\overline{p}}$ and negative $\overline{p}$. When a cast error arises during execution, the blame's polarity indicates whether it is the cast expression that violates the typing assertions of the cast (positive blame) or the context (negative blame).

Figure 2 contains the dynamics of the calculus. Base type operations are unsurprising (BA-SG-Op). They just switch to the new algebras. For functions, β_v reduction is kept unchanged (BA-S-BetaV). It remains to consider casts.

The base type cast BA-SG-Cast-Base checks the annotation and converts between their **S** and **D** shapes while keeping the underlying annotation a. The relation $\prec$ expresses compatibility of a value annotation with a type annotation. Any dynamic value annotation is compatible with the type annotation ? and a static value annotation of the form $\mathbf{S}(a)$ is compatible with a.

$$\mathbf{D}(a) \prec ? \qquad \mathbf{S}(a) \prec a$$

Type casts at non-base types are treated by decomposing the cast into its constituent casts and distributing them according to the type constructor, exemplified with casting of values of function type BA-SG-Cast-Fun. Due to the contravariance of the function type, the polarity of the blame on the function argument flips but the polarity on the function result remains the same.

With respect to λ^{BA}, the typing rule for operations changes and the rule for casts gets added. Even the rule for operations just switches the handling of the annotations to the algebra $\mathcal{A}^?$. The rule for constants needs to be slightly adjusted to require the compatibility of annotations.

$$\frac{\Gamma \vdash_G e_1 : B[a_1] \qquad \Gamma \vdash_G e_2 : B[a_2] \qquad (a_1 \oplus_{A^?} a_2) =: a}{\Gamma \vdash_G e_1 \oplus e_2 : B[a]}\ \text{BA-TG-Op}$$

$$\frac{V(a) \prec ta}{\Gamma \vdash b[V(a)] : ta}\ \text{BA-TG-Const}$$

The typing rule for casts enforces that casts are only executed for compatible annotated types as indicated by a compatibility relation.

$$\frac{\Gamma \vdash_G e : t_1 \qquad t_1 \sim t_2}{\Gamma \vdash_G (e : t_1 \rightsquigarrow^p t_2) : t_2}\ \text{BA-TG-Cast}$$

$$B[?] \sim B[ta] \qquad B[ta] \sim B[?] \qquad B[ta] \sim B[ta] \qquad \frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{t_1 \to t_2 \sim t_1' \to t_2'}$$

Fig. 3. Compatibility

The compatibility relation $\sim$ (Figure 3) ensures that two types have the same underlying structure and that direct casts between statically annotated types are ruled out. This relation is reflexive and symmetric, but not transitive. For a transitive compatibility, $B[a] \sim B[?]$ and $B[?] \sim B[a']$ would imply $B[a] \sim B[a']$ if $a \neq a'$. Using the annotation algebra for dimensions, such a cast could try to convert metres to seconds (and would always fail). Intransitive compatibility makes it harder to write obviously faulty code by only allowing casts between static and dynamic annotations. Furthermore, if $B[a] \sim B[a']$ for $a \neq a'$, then the BA-SG-Cast-Base rule would fail on a static cast that should be disallowed by the type system. Also, the formulation of the technical results in Section 3.3 would get more complicated (particularly Definition 1).

3.3 Results

We have established type soundness for the gradual calculus. The most interesting part of the result is the progress lemma because it comes with a characterization of the possibly failing terms, the dynamically stuck terms.

Definition 1. *A term e is* dynamically stuck *if*

1. $e = E[b[\boldsymbol{D}(a)] : B[?] \rightsquigarrow^p B[a']]$ *where* $a \neq a'$,
2. $e = E[b_1[V_1(a_1)] \oplus b_2[V_2(a_2)]]$ *where* $(a_1, a_2) \notin dom(\oplus_A)$ *and* $V_i = \boldsymbol{D}$ *for some* $i \in \{1, 2\}$,
3. $e = E[b_1[V_1(a_1)] \oplus b_2[V_2(a_2)]]$ *where* $(a_1, a_2) \in dom(\oplus_A)$ *but* $(b_1, b_2) \notin dom(\oplus_B)$.

The core reason for being dynamically stuck is either a failing cast of a dynamically annotated value to a statically annotated one, where the provided annotation is not the expected one, or a failing attempt at a dynamically checked operation. For the failing cast, we also say that it *raises blame p* according to the blame label attached to the cast. A third case arises when $\oplus_B$ is partial, but its occurrence depends on the abstraction implemented by the annotation algebra. It is thus independent of gradual typing.

Lemma 1 (Progress). *If* $\cdot \vdash_G e : t$ *then either e is a value or* $(\exists e')\ e \longrightarrow e'$ *or e is a dynamically stuck term.*

Lemma 2 (Preservation).
If $\cdot \vdash_G e : t$ *and* $e \longrightarrow e'$, *then* $\cdot \vdash_G e' : t$.

$$B[ta] <:^{\circ} B[ta] \qquad B[ta] <:^{+} B[ta]$$

$$B[a] <:^{\circ} B[?] \qquad B[a] <:^{+} B[?] \qquad B[ta_1] <:^{-} B[ta_2]$$

$$\frac{t_2' <:^{\circ} t_1' \qquad t_1 <:^{\circ} t_2}{t_1' \to t_1 <:^{\circ} t_2' \to t_2} \qquad \frac{t_2' <:^{+} t_1' \qquad t_1 <:^{-} t_2}{t_1' \to t_1 <:^{-} t_2' \to t_2} \qquad \frac{t_2' <:^{-} t_1' \qquad t_1 <:^{+} t_2}{t_1' \to t_1 <:^{+} t_2' \to t_2}$$

Fig. 4. Cast-related subtyping relations

Following Wadler and Findler's Blame Calculus [32], the subsequent development works towards a blame theorem for λ_G^{BA}. The blame theorem is a sharpening of progress which further examines the nature of the casts [31,32]. It gives sufficient conditions on casts to ensure that all blame falls on the dynamically checked parts of the program.

Casts are classified according to a number of subtyping relations which are not meant to be used for subsumption: plain subtyping, positive subtyping, and negative subtyping. Figure 4 defines them for gradual annotated typing.

Plain subtyping classifies casts that perform safe conversions and thus never cause a run-time error: A cast from t to t' is safe if $t <:^{\circ} t'$. Intuitively, casts are safe if they are trivial, or inject statically checked expressions into dynamic code. In the latter case, the dynamic code has the complete freedom and responsibility to use the statically typed results adequately. Examples of such safe injections are $e : B[a] \rightsquigarrow^{p} B[?]$, or $e : B[a] \to B[a] \rightsquigarrow^{p} B[a] \to B[?]$. Trying to inject a dynamic value into static code (e.g. $e : B[?] \rightsquigarrow^{p} B[a]$) could result in a run-time error, which is unsafe. Plain subtyping on base-types allows the identity conversion and a conversion from a static annotation to a dynamic one. Blame subtyping for function types is contravariant in the parameter type. A cast like $e : B[?] \to B[a] \rightsquigarrow^{p} B[a] \to B[a]$ is considered safe because it relies on the function's original type which already claims full responsibility for the parameter.

As in Wadler and Findler's work, plain subtyping may be factored into positive subtyping $<:^{+}$ and negative subtyping $<:^{-}$. If positive (negative) subtyping $t_1 <:^{+(-)} t_2$ holds then a cast from t_1 to t_2 with label p does not result in a run-time error that raises blame p ($\overline{p}$). Positive subtyping is analogous to plain subtyping on base types but relaxes the restriction on function parameters to *negative* subtyping. Negative subtyping only restricts type annotations for function parameters (via positive subtyping), as only function casts may invert blame labels.

Lemma 3. *The relations $<:^{+}$, $<:^{-}$, and $<:^{\circ}$ are reflexive and transitive.*

Further, plain subtyping is the intersection of positive and negative subtyping.

Lemma 4. $<:^{\circ} = <:^{+} \cap <:^{-}$.

$$x \text{ sf } p \qquad \frac{e_1 \text{ sf } p \qquad e_2 \text{ sf } p}{e_1\, e_2 \text{ sf } p} \qquad \frac{e \text{ sf } p}{\lambda x.e \text{ sf } p} \qquad b[va] \text{ sf } p \qquad \frac{e_1 \text{ sf } p \qquad e_2 \text{ sf } p}{e_1 \oplus e_2 \text{ sf } p}$$

$$\frac{e \text{ sf } p \qquad q \notin \{p, \overline{p}\}}{e : t_1 \rightsquigarrow^q t_2 \text{ sf } p} \qquad \frac{e \text{ sf } p \qquad t_1 <:^+ t_2}{e : t_1 \rightsquigarrow^p t_2 \text{ sf } p} \qquad \frac{e \text{ sf } p \qquad t_1 <:^- t_2}{e : t_1 \rightsquigarrow^{\overline{p}} t_2 \text{ sf } p}$$

Fig. 5. Safety with respect to p

The next step towards showing a blame theorem consists of defining a set of expressions that is safe for a certain blame label p. The judgment e sf p in Figure 5 characterizes this set. It guarantees that all cast operations in e that involve the label p (or $\overline{p}$) use types that are related by positive (negative) subtyping. Fortunately, safety is an invariant under reduction.

Lemma 5. *If e sf p and $e \longrightarrow e'$, then e' sf p.*

The blame theorem states that an irreducible term, which is safe for p, cannot be stuck on a cast labeled p. An irreducible term, which is safe for p and $\overline{p}$, cannot be stuck on a cast labeled p or $\overline{p}$.

Lemma 6. *If e sf p and $\neg(\exists e')\ e \longrightarrow e'$, then e cannot have the form $E[b[\boldsymbol{D}(a)] : B[?] \rightsquigarrow^p B[a']]$, where $a \neq a'$.*

Theorem 1 (Blame). *If e sf p and e sf $\overline{p}$ and $\neg(\exists e')\ e \longrightarrow e'$, then e cannot have the form $E[b[\boldsymbol{D}(a)] : B[?] \rightsquigarrow^p B[a']]$ or $E[b[\boldsymbol{D}(a)] : B[?] \rightsquigarrow^{\overline{p}} B[a']]$, where $a \neq a'$.*

3.4 Subtyping

A reflexive and transitive conversion relation $\precsim$ on the annotation algebra for base types induces a subtyping relation on the corresponding annotated type system. The required subsumption rule is standard.

$$\frac{a_1 \precsim a_2}{B[a_1] <: B[a_2]} \qquad \frac{t_1' <: t_1 \qquad t_2 <: t_2'}{t_1 \to t_2 <: t_1' \to t_2'} \qquad \frac{\Gamma \vdash e : t \qquad t <: t'}{\Gamma \vdash e : t'}$$

In the presence of conversion, the static annotation on a value need no longer be equal to the static annotation on its type. Hence, the type-level operation $\oplus_{A^?}$ and the value-level operation $\oplus_A$ may yield different results because they are applied to different arguments, albeit related by $\precsim$. This observation leads to the requirement that $\oplus_A$ must be monotonic with respect to $\precsim$. In particular, if $a_1 \precsim a_1'$, $a_2 \precsim a_2'$, and $a_1' \oplus_A a_2' =: a'$, then $a_1 \oplus_A a_2 =: a$ and $a \precsim a'$. Otherwise, reduction may get stuck on a well-typed term and type preservation may fail.

To see that, consider $b_i[a_i] : B[a_i']$ where $a_i \precsim a_i'$ $(i = 1, 2)$. If $a_1' \oplus_A a_2' =: a'$, then the term $b_1[a_1] \oplus b_2[a_2] : B[a']$ is well-typed. However, the reduction of $\oplus$ gets stuck unless $a_1 \oplus_A a_2 =: a$ holds and type preservation fails unless $a \precsim a'$.

For the gradual system, the subtyping relation needs to be extended to ? annotations. They are not convertible with any other annotation so that annotations cannot become dynamic (and vice versa) without an explicit cast.

$$B[?] <:_G B[?]$$

In particular, having $B[a] \not<:_G B[?]$ prevents the unintentional introduction of dynamic values. The remaining cases are as in the static system.

Nothing else needs to change, except that the compatibility relation between value annotations and type annotations that is used in the static checking of casts (Figure 2) has to reflect the possible conversion.

$$\frac{a \precsim a'}{\mathbf{S}(a) \prec a'}$$

In the presence of subtyping, one might contemplate to slacken the compatibility relation $\sim$ and admit the cast between annotations that are related by subtyping. That is, the axiom $B[ta] \sim B[ta]$ would be refined to $B[ta] \sim B[ta']$ if $ta \precsim ta'$ or $ta' \precsim ta$. However, this refined axiom introduces the danger that casts that do not involve ? may raise blame at run time: Each downcast involves a run-time check. The blame theorem can be refined to distinguish safe upcasts and unsafe downcasts by including the conversion relation in the blame subtyping of static base-types: $B[a] <:^\circ B[a']$ and $B[a] <:^+ B[a']$ whenever $a \precsim a'$.

3.5 Alternative Modeling

A notion of gradual typing could also be introduced without an extended algebra, just with the plain annotation algebra $\mathcal{A}$ for value annotations. The compatibility relation between value annotations and type annotations would relate any value annotation to ? and otherwise be the equality on plain annotations:

$$a \prec' ? \qquad a \prec' a$$

With this change, the cast operation (as in BA-SG-Cast-Base) would never modify any annotation. The dynamics of operations would correspond to BA-S-Op. The definition of dynamically stuck terms (Definition 1) would change as follows.

Definition 2. *A term e is* stuck *if*

1. $e = E[b[a] : B[?] \rightsquigarrow^p B[a']]$ *where* $a \neq a'$,
2. $e = E[b_1[a_1] \oplus b_2[a_2]]$ *where* $(a_1, a_2) \notin dom(\oplus_A)$,
3. $e = E[b_1[a_1] \oplus b_2[a_2]]$ *where* $(a_1, a_2) \in dom(\oplus_A)$ *but* $(b_1, b_2) \notin dom(\oplus_B)$.

Comparing Definitions 1 and 2 shows that the plain annotation algebra weakens the progress result. While cases 1 and 3 yield the same information as cases 1 and 3 in Definition 1, case 2 has become ambiguous: In case 2 of Definition 1 it is clear that the annotation mismatch is caused by an attempt to apply $\oplus$ in a dynamically typed fragment. With Definition 2 the annotation mismatch can no longer be located; it might be in a statically typed part of the program.

We conclude that the simplified approach is weaker than the $\mathcal{A}^+$-approach presented in Section 3.2 because it yields a less informative progress result that only makes an ambiguous statement about a key part of the type system.

3.6 Annotation Erasure

Using the $\mathcal{A}^+$-approach, we may define an erasure translation that avoids the passing of annotations at run time in the statically checked parts of a program. In the target calculus of this translation, the cast operations amount to adding or removing run-time annotations. The syntax of this calculus extends the syntax for base type refinements with unannotated base-type values:

$$e ::= b \mid \ldots \qquad v ::= b \mid \ldots$$

The erasure translation $|\cdot|$ only acts on annotated base-type values and extends homomorphically to the remaining syntactic constructs:

$$|b[\mathbf{S}(a)]| = b \qquad |b[\mathbf{D}(a)]| = b[a]$$

Besides rule BA-S-Op, there is an additional computation rule for unannotated base-type values:

$$\text{BA-S-Op}\ \frac{b_1 \oplus_B b_2 =: b \qquad a_1 \oplus_A a_2 =: a}{b_1[a_1] \oplus b_2[a_2] \longrightarrow' b[a]} \qquad \text{BA-SG-Op'}\ \frac{b_1 \oplus_B b_2 =: b}{b_1 \oplus b_2 \longrightarrow' b}$$

Reduction of base type casts is best presented as three separate rules.

$$\text{BA-SG-Cast-Trivial}\ (v : B[ta] \rightsquigarrow^p B[ta]) \longrightarrow' v \qquad \text{BA-SG-Cast-ToDyn}\ (b : B[a] \rightsquigarrow^p B[?]) \longrightarrow' b[a]$$

$$\text{BA-SG-Cast-FromDyn}\ (b[a] : B[?] \rightsquigarrow^p B[a]) \longrightarrow' b$$

Trivial casts are discarded. A cast from a static type into a dynamic one adds the annotation of the static type as a run-time annotation. A cast from dynamic to static strips off the run-time annotation, provided it matches that of the static destination type.

Progress for this calculus needs yet another notion of stuck terms.

Definition 3. *A term e is* stuck *if*

1. $e = E[b[a] : B[?] \rightsquigarrow^p B[a']]$ *where* $a \neq a'$,
2. $e = E[b_1[a_1] \oplus b_2[a_2]]$ *where* $(a_1, a_2) \notin dom(\oplus_A)$,
3. $e = E[b_1 \oplus b_2]$ *where* $(b_1, b_2) \notin dom(\oplus_B)$.

This definition is again unsatisfactory. The first and second cases correspond to Definition 1. However, in the third case, computations with unannotated base-type values never check their annotation. Hence, the condition imposed by the static typing rule for primitive operations does not correspond to a run-time restriction, which trivializes preservation and progress.

We conclude that this calculus is also unsuitable to prove a strong progress result and we see that as a further indication in favor of the $\mathcal{A}^+$-approach.

However, it is possible to relate the $\mathcal{A}^+$-approach with the erasure approach, which amounts to an efficient implementation. For typed expressions, the evaluation relations $\longrightarrow$ and $\longrightarrow'$ simulate each other in lockstep.

Lemma 7. *Let e be a closed expression.*

1. *If $e \longrightarrow e'$, then $|e| \longrightarrow' |e'|$.*
2. *If $\cdot \vdash_G e : t$ and $|e| \longrightarrow' e''$, then $e \longrightarrow e'$ and $e'' = |e'|$.*

As an example that typing is essential for item 2 in the lemma, consider the expression $1[S(m)] + 1[S(kg)]$ in the calculus for dimensions. It is not typeable and it is stuck at rule BA-SG-Op. However, its erasure $|1[S(m)]+1[S(kg)]| = 1+1$ reduces to 2 using $\longrightarrow'$.

4 Eliminating Run-Time Checks

A gradually typed program with manually inserted casts can be improved by a type-preserving transformation, $e \Longrightarrow e'$, that increases the amount of statically handled annotations and decreases the number of dynamic checks without eliminating potential annotation mismatches. Thus, the transformed program should be equivalent to the original one, but with less dynamic annotation handling.

To express the results of the transformation concisely, we introduce a new kind of term $e ::= (\!|p|\!) \mid \ldots$ where $(\!|p|\!)$ is an *exception package* that carries blame label p. An exception package is generated by failing cast expressions, it is propagated upwards through evaluation contexts, and it has any type.

$$\frac{a \neq a'}{b[\mathbf{D}(a)] : B[?] \rightsquigarrow^p B[a'] \longrightarrow (\!|p|\!)} \qquad \frac{e \longrightarrow (\!|p|\!)}{E[e] \longrightarrow (\!|p|\!)} \qquad \Gamma \vdash_G (\!|p|\!) : t$$

4.1 Transformation Rules

In a typed term, any cast may be executed on a base type constant b. The result e' is either the same value b with a different annotation or an exception $(\!|p|\!)$. Exceptions may be promoted across evaluation contexts.

$$\text{BA-TR-Const}\ \frac{b[va] : B[ta] \rightsquigarrow^p B[ta'] \longrightarrow e'}{b[va] : B[ta] \rightsquigarrow^p B[ta'] \Longrightarrow e'} \qquad \text{BA-TR-Blame}\ E[(\!|p|\!)] \Longrightarrow (\!|p|\!)$$

If a cast is applied to a dynamic operation $\oplus$, then the annotations of the arguments can be determined from the annotation of the result if $\oplus_A$ is locally injective. Only in this case, the cast can be propagated to the arguments.

$$\text{BA-TR-Op}\ \frac{\oplus_A^{-1}(a) = \{(a_1, a_2)\}}{e_1 \oplus e_2 : B[?] \rightsquigarrow^p B[a] \Longrightarrow (e_1 : B[?] \rightsquigarrow^p B[a_1]) \oplus (e_2 : B[?] \rightsquigarrow^p B[a_2])}$$

The blame annotation gets propagated to the arguments to preserve the error messages. A typical example where this rule is applicable is an addition expression in the system for dimensions, where the annotations of both arguments are equal to the annotation of the result.

If a cast is applied to a lambda expression, then the expression can be transformed analogously to the dynamics for the cast (rule BA-SG-Cast-Fun). However, the cast on the result is pushed inside to the body of the lambda to be able to continue the transformation.

BA-TR-Fun
$$(\lambda x.e) : (t_1 \to t_2) \rightsquigarrow^p (t_1' \to t_2') \Longrightarrow \lambda x.(\lambda x.(e : t_2 \rightsquigarrow^p t_2'))(x : t_1' \rightsquigarrow^{\overline{p}} t_1)$$

Any cast applied to a function application may be pushed towards the function, which might enable rule BA-TR-Fun.

BA-TR-App
$$\frac{e_2 : t_2}{(e_1\, e_2) : t \rightsquigarrow^p t' \Longrightarrow (e_1 : t_2 \to t \rightsquigarrow^p t_2 \to t')\, e_2}$$

The interplay between BA-TR-App and BA-TR-Fun may generate identity casts, which may safely be omitted.

BA-TR-Id
$$e : t \rightsquigarrow^p t \Longrightarrow e$$

It also makes sense to consider transforming casts nested in elimination positions. If both operands of an operation are (positive) casts to dynamic, then these casts can be merged and propagated to the result. The blame labels need not be preserved because a positive cast on a base type never fails. The transformation arbitrarily chooses the left operand's blame label.

BA-TR-Op-Elim
$$\frac{a_1 \oplus_A a_2 =: a}{(e_1 : B[a_1] \rightsquigarrow^{p_1} B[?]) \oplus (e_2 : B[a_2] \rightsquigarrow^{p_2} B[?]) \Longrightarrow e_1 \oplus e_2 : B[a] \rightsquigarrow^{p_1} B[?]}$$

We may also state rules for lifting casts out of function bodies and out of function applications. However, the overall approach of our transformation is to start at the root of a term and to push casts as far inside as possible. This approach does not require such lifting rules. Applying our rules exhaustively in a top-down manner results in a term where each casts is either applied to a variable, to an application of a primitive operation, or to another cast. However, we stress that each transformation step is correct in any context.

To further optimize the resulting term additionally requires an approach for merging two casts into one. A few special cases of this merge can be stated easily. However, a satisfactory treatment of cast composition requires a different representation of casts and a careful consideration of blame propagation. There are at least two alternatives for this representation, either threesomes [28] or coercions [11], but their introduction is not in scope of this paper.

4.2 Contextual Equivalence and Bisimulation

To prove the correctness of the transformation rules, we establish that they are contextual equivalences in λ_G^{BA}. Two expressions e_1 and e_2 are contextually equivalent if they behave the same in every context [19].

Now, for all contexts C, if

1. there exists v such that $C[e_1] \longrightarrow^* v$ and $C[e_2] \longrightarrow^* v$, or
2. there exists p such that $C[e_1] \longrightarrow^* (\!|p|\!)$ and $C[e_2] \longrightarrow^* (\!|p|\!)$, or
3. $C[e_1] \Uparrow$ iff $C[e_2] \Uparrow$[2]

then e_1 and e_2 are contextually equivalent, written $e_1 \simeq e_2$.

As contextual equivalence is hard to prove directly, we prove it via bisimulation. To this end, we define a notion of observations $\alpha ::= @v \mid b[a] \mid (\!|p|\!)$ for λ_G^{BA} programs. An observation on a function type is the application of a value $@v$. On a base type, we may observe the annotated base-type value $b[a]$. On a failing computation, we observe the blame label raised $(\!|p|\!)$. This observation is possible at any type.

Based on this notion of observations, we define a labeled transition system. Basic values are emitted as observations and their transition yields a non-terminating expression $\mathbf{0}$ with an empty derivation tree. Blame exceptions are treated in the same way. At function type, a transition is only possible on a function which is applied to a value of suitable type. This treatment is an adaptation of the call-by-value variation of Gordon's applicative bisimulation theory [9].

$$\text{LT-Base}\quad b[a] \overset{b[a]}{\Longmapsto} \mathbf{0} \qquad \text{LT-Blame}\quad (\!|p|\!) \overset{(\!|p|\!)}{\Longmapsto} \mathbf{0}$$

$$\text{LT-App}\quad \frac{\cdot \vdash_G v : t' \to t \qquad \cdot \vdash_G v' : t}{v \overset{@v'}{\Longmapsto} v\, v'} \qquad \text{LT-Comp}\quad \frac{e \longrightarrow e' \qquad e' \overset{\alpha}{\Longmapsto} e''}{e \overset{\alpha}{\Longmapsto} e''}$$

This definition of the labeled transition relation is adequate because an expression can make a transition if and only if it either terminates in a value or in a blame exception.

Lemma 8. *$e \overset{\alpha}{\Longmapsto} e'$ iff either there exists v such that $e \longrightarrow^* v$ or there exists p such that $e \longrightarrow^* (\!|p|\!)$.*

From the labeled transition system, we define bisimilarity as usual. For a relation $\mathcal{S} \subseteq \mathit{Exp} \times \mathit{Exp}$, define two functions:

$$[\mathcal{S}] := \{(e_1, e_2) \mid (\exists \alpha, e_1')\ e_1 \overset{\alpha}{\Longmapsto} e_1' \Rightarrow (\exists e_2')\ e_2 \overset{\alpha}{\Longmapsto} e_2', e_1' \mathcal{S} e_2'\}$$
$$\langle \mathcal{S} \rangle := [\mathcal{S}] \cap [\mathcal{S}^{op}]^{op}$$

Here, $\mathcal{S}^{op} = \{(e_2, e_1) \mid (e_1, e_2) \in \mathcal{S}\}$ is the opposite relation to $\mathcal{S}$. Both functions are easily checked to be monotone, so we can take their greatest fixpoint $\sim\ = \nu\mathcal{S}.\langle \mathcal{S} \rangle$, which is the bisimilarity relation for the calculus λ_G^{BA}.

Lemma 9.

1. *$\sim$ is an equivalence relation.*
2. *Evaluation steps are bisimilar: $\longrightarrow\ \subseteq\ \sim$.*

[2] $e \Uparrow$ if for each e' such that $e \longrightarrow^* e'$ there exists e'' such that $e' \longrightarrow e''$.

Using Gordon's adaptation [9] of Howe's method [12], it can be shown that bisimilarity is a congruence and that it coincides with contextual equivalence.

The soundness of the transformation rules is proven by strong coinduction. As a corollary, we obtain that each transformation rule is a contextual equivalence in λ_G^{BA}.

5 Related Work

Disney and Flanagan [5] have applied gradual typing to a type system for information flow. They have proved type safety, noninterference, and a blame theorem. Their approach is tailored to the particular system. They do not systematically transform the annotation strategy of an existing system, but define notions like positive and negative subtyping directly on the existing annotations.

Gradual typing has made an impact on object-oriented language design, so the following related work comes from that area. As our method is mainly geared towards functional programming, it is not directly applicable. In addition, each of the related work items addresses a very specific point for gradualization that is deeply intertwined with the rest of the language design considered.

Typestate is a refinement of an (imperative object) type that changes as a program progresses. There are distinguished operations that change the typestate and the typestate governs which operations are available. Gradual typestate by Wolff and coworkers [33] addresses the problem that a program with typestate requires extensive type annotations to indicate the typestate transitions (e.g., on function arguments) and to manage sharing: if a function obtains two aliases to a typestate object, then applying different operations to them may lead to unsoundness. The gradual version of the system allows to replace typestate-related annotations by `Dyn` and performs the corresponding state and permission checks at run time. The authors prove type soundness and define a semantics by translating to a lower level calculus. The construction of the gradual extension is closely tied to the particulars of the system, in particular with the handling of aliasing. The aliasing aspect is not considered in our work.

The goal of gradual ownership types by Sergey and Clarke [24], also from the realm of object-oriented programming, is to enable a smooth migration from systems without control of ownership to static control of ownership. They apply the ideas of gradual typing to express heap properties instead of properties of values or (single) objects. One motivation is in avoiding the excessive annotation overhead that comes with other ownership type systems. In the construction of the system, the particular dynamic enforcement mechanism is an ad-hoc design. In this system, assignments have to be checked in order to prevent unwanted paths in the object graph.

Ina and Igarashi have considered gradual typing for generics [13]. Their system extends gradual typing to an object-oriented language with bounded subtyping. As such it discusses an extension to parameterized types, not to annotated. Thus, the required dynamic checks are tests on the run-time type of an object, not for additional properties.

Tobin-Hochstadt and Felleisen [31] were the first to investigate the boundaries of static and dynamic checking with a blame theorem. Wadler and Findler's subsequent analysis of blame [32] has been a major source of inspiration. We managed to transfer their results to a wide range of annotated type systems. Hybrid type checking [17] is a system based on dependent types and base-type refinements, which are described by arbitrary predicates. Dynamic checks (casts) serve to manifest refinements in types where they can be exploited in static checking. Here, the dynamically annotated type is essentially one where the predicate is true. Subtyping needs to be checked with a theorem prover. In contrast, our approach is geared toward refining types with additional properties that are not just predicates on the values of a base type.

The transformation of programs with coercions has been considered by Henglein [11]. Different to the discussion in our paper, his calculus employs coercion expressions that are built from primitive coercions on base types using functorial operations. Furthermore, he develops an equational theory of coercions and of expressions with coercions. His theory is not directly linked with contextual equivalence.

A coercion calculus with blame has been investigated by Siek, Garcia, and Taha [26]. They consider design alternatives for higher-order casts, where they also analyze the problem of merging two casts at run time. One of their options is to fail early, when casts are composed. In a program transformation as we consider it in Section 4, such a behavior would yield false positives when processing dead code.

Siek and Wadler [28] discuss a variation of the blame calculus where arbitrarily long compositions of casts are compressed into a single, equivalent threesome cast. They also show the equivalence of threesomes and a normalizing coercion calculus. A mechanism like that should be integrated in our simplifying transformation. We leave it to future work because it would require a reworking of the annotated cast mechanism, either in terms of coercions or in terms of threesomes.

Rastogi and coworkers [22] develop an algorithm for type inference in ActionScript that also aims to eliminate run-time checks, similar to our transformation. Their approach is not based on program transformation. Instead, their algorithm replaces the dynamic type with type variables and globally computes and solves set constraints that overapproximate the flows of types to contexts. The algorithm preserves run-time errors with respect to the original untyped program but sacrifices blame guarantees for improved precision. In contrast, our transformation preserves errors and blame as each transformation preserves contextual equivalence.

6 Conclusion

We show that annotated type systems, where the annotation is restricted to base types, can be gradualized by applying a simple procedure. The core of our approach is the definition of a generic gradual annotated type system based on an annotation algebra. We demonstrate its applicability by instantiating it to

several examples. The technical results (type soundness and blame theorems) for the generic gradual systems have generic reusable proofs that can be instantiated to each annotation algebra.

Specific semantic properties still require extra work: the generic results hold, even if the dynamic manipulation of the annotations is total nonsense. For example, a sound gradual security type system requires that the handling of dynamic annotations guarantees noninterference, but the specifics are not prescribed by our framework.

Our type system can be extended in several directions. Annotations may be added to each type constructor: this extension is necessary for an information flow analysis that can guarantee noninterference. The calculus may be extended with annotation polymorphism, which diminishes the need for the dynamic handling of annotations. Also the whole system may be based on a calculus with ML-style polymorphism.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. ACM TOPLAS 28(2), 207–255 (2006)
2. Castagna, G. (ed.): ESOP 2009. LNCS, vol. 5502. Springer, Heidelberg (2009)
3. Chin, B., Markstrum, S., Adsul, B.: Inference of user-defined type qualifiers and qualifier rules. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 264–278. Springer, Heidelberg (2006)
4. Darwin, I.F.: Annabot: A static verifier for java annotation usage. Adv. Software Engineering (2010)
5. Disney, T., Flanagan, C.: Gradual information flow typing. In: STOP 2011 (2011)
6. Fennell, L., Thiemann, P.: Gradual security typing with references. In: Cortier, V., Datta, A. (eds.) CSF, pp. 224–239. IEEE (2013)
7. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 219–232. ACM Press (June 2000); 35(5) of SIGPLAN Notices
8. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proc. PLDI 1991, pp. 268–277. ACM (June 1991)
9. Gordon, A.D.: Bisimilarity as a theory of functional programming. Theoretical Computer Science 228(1-2), 5–47 (1999)
10. Heintze, N., Riecke, J.G.: The SLam calculus: Programming with security and integrity. In: Cardelli, L. (ed.) Proc. 25th ACM Symp. POPL, pp. 365–377. ACM Press (January 1998)
11. Henglein, F.: Dynamic typing: Syntax and proof theory. Science of Computer Programming 22, 197–230 (1994)
12. Howe, D.: Proving congruence of bisimulation in functional programming languages. Information and Computation 124(2), 103–112 (1996)
13. Ina, L., Igarashi, A.: Gradual typing for generics. In: Lopes, C.V., Fisher, K. (eds.) OOPSLA, pp. 609–624. ACM (2011)
14. Jackson, D.: Aspect: Detecting bugs with abstract dependences. ACM Trans. Softw. Eng. Methodol. 4(2), 109–145 (1995)

15. Kennedy, A.: Dimension types. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788, pp. 348–362. Springer, Heidelberg (1994)
16. Kennedy, A.J.: Relational parametricity and units of measure. In: Jones, N. (ed.) Proc. 1997 ACM Symp. POPL, pp. 442–455. ACM (January 1997)
17. Knowles, K.L., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. 32(2) (2010)
18. Leijen, D., Meijer, E.: Domain-specific embedded compilers. In: 2nd Conference on Domain-Specific Languages. USENIX (October 1999), `http://usenix.org/events/dsl99/index.html`
19. Morris Jr., J.H.: Lambda Calculus Models of Programming Languages. PhD thesis. MIT Press (December 1968)
20. Nielson, F.: Annotated type and effect systems. Computing Surveys 28(2), 344–345 (1996)
21. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. In: Aiken, A. (ed.) Proc. 26th ACM Symp. POPL, pp. 276–290. ACM Press (January 1999)
22. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. In: Proc. 39th ACM Symp. POPL, pp. 481–494. ACM Press (January 2012)
23. Rittri, M.: Dimension inference under polymorphic recursion. In: Peyton Jones, S. (ed.) Proc. FPCA 1995, pp. 147–159. ACM (June 1995)
24. Sergey, I., Clarke, D.: Gradual ownership types. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 579–599. Springer, Heidelberg (2012)
25. Siek, J.G., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007)
26. Siek, J.G., Garcia, R., Taha, W.: Exploring the design space of higher-order casts. In: Castagna [2], pp. 17–31
27. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
28. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Palsberg, J. (ed.) Proc. 37th ACM Symp. POPL, pp. 365–376. ACM Press (January 2010)
29. Solberg, K.L.: Annotated Type Systems for Program Analysis. PhD thesis, Odense University, Denmark. Also technical report DAIMI PB-498, Comp. Sci. Dept. Aarhus University (July 1995)
30. Tang, D., Plsek, A., Vitek, J.: Static checking of safety critical Java annotations. In: Kalibera, T., Vitek, J. (eds.) JTRES. ACM International Conference Proceeding Series, pp. 148–154. ACM (August 2010)
31. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Dynamic Languages Symposium, DLS 2006, pp. 964–974. ACM (2006)
32. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna [2], pp. 1–16
33. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual typestate. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 459–483. Springer, Heidelberg (2011)