Well-Typed Programs Can't Be Blamed

Philip Wadler[1] and Robert Bruce Findler[2]

[1] University of Edinburgh
[2] University of Chicago

Abstract. We introduce the *blame calculus*, which adds the notion of blame from Findler and Felleisen's *contracts* to a system similar to Siek and Taha's *gradual types* and Flanagan's *hybrid types*. We characterise where positive and negative blame can arise by decomposing the usual notion of subtype into positive and negative subtypes, and show that these recombine to yield naive subtypes. Naive subtypes previously appeared in type systems that are unsound, but we believe this is the first time naive subtypes play a role in establishing type soundness.

1 Introduction

Much recent work has focused on integrating dynamic and static typing using *contracts* [4] to ensure that dynamically-typed code meets statically-typed invariants. Examples include *gradual types* [15], *hybrid types* [5, 8], *dynamic dependent types* [13], and *multi-language programming* [10]. Both Meijer [11] and Bracha [2] argue in favor of mixing dynamic and static typing. Static and dynamic typing are both supported in Visual Basic, and similar integration is planned for Perl 6 and ECMAScript 4.

Here we unify some of this work, by introducing a notion of blame (from contracts) into a type system with casts (similar to intermediate languages for gradual and hybrid types), yielding a system we dub the *blame calculus*. In this calculus, programmers may add casts to evolve dynamically typed code into statically typed, (as with gradual types) or to evolve statically typed code to use subset types (as with hybrid types).

The technical content of this paper is to introduce notions of positive and negative subtypes, and prove a theorem that characterises when positive and negative blame can occur. A corollary is that when a program integrating less-typed and more-typed components goes wrong the blame must lie with the less-typed component. Though obvious, this result has either been ignored in previous work or required a complex proof; here we give a simple proof.

Our work involves both ordinary subtypes (which, for functions, is contravariant in the domain and covariant in the range) and naive subtypes (which is covariant in both the domain and the range). Ordinary subtypes characterize a cast that cannot fail, while naive subtypes characterize which side of a cast is less typed (and hence will be blamed if the cast fails). We show that ordinary subtypes decompose into positive and negative subtypes, and that these recombine in a different way to yield naive subtypes. A striking analogy is a tangram, where a square decomposes into parts that recombine into a different shape (see Figure 1). Naive subtypes previously appeared in type systems that are unsound, notably that of Eiffel [12], but we believe this is the first time naive subtypes play a role in establishing type soundness.

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 1–16, 2009.

Gradual types [15], hybrid types [5, 8], and dynamic dependent types [13] use source languages where most or all casts are omitted, but inferred by a type-directed translation; all three use similar translations which target similar intermediate languages. The blame calculus resembles these intermediate languages. Our point is that the intermediate language is in itself suitable as a source language, with the advantage that it becomes crystal clear where static guarantees hold and where dynamic checks are enforced.

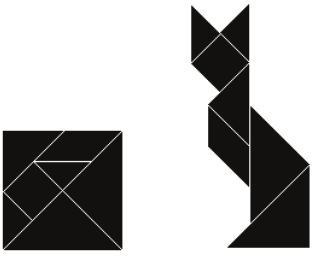

Fig. 1. Tangram as metaphor: Ordinary subtyping decomposes and recombines to yield naive subtyping

The blame calculus uses subset types as found in hybrid types and dynamic dependent types, but it lacks the dependent function types found in these systems (an important area for future work). Hybrid types and dynamic dependent types are parameterized by a theorem prover, which returns true, false, or maybe when supplied with a logical implication required by a subtyping relationship; the blame calculus corresponds to the extreme case where the theorem prover always returns maybe.

We make the following contributions.

- We introduce the blame calculus, showing that a language with explicit casts is suited to many of the same purposes as gradual types and hybrid types (Section 2).
- We give a framework similar to that of hybrid types and dynamic dependent types, but with a decidable type system for the source language (Section 3).
- We factor ordinary subtypes positive and negative subtypes, which recombine into naive subtypes. We prove that a cast from a positive subtype cannot give rise to positive blame, and that a cast from a negative subtype cannot give rise to negative blame (Section 4).

An earlier version of this paper appeared in a workshop [19]. The current version is completely rewritten and some technicalities differ. A rule merging positive blame and negative blame from distinct casts has been eliminated, and as a consequence we use a simpler notation with one label rather than two. A rule making every ground type a subtype of type `Dyn` has been added, making the subtyping relations less conservative. Detailed proofs may be found in the accompanying technical report [20].

2 The Blame Calculus

2.1 From Untyped to Typed

Figure 2 presents a series of example programs, which we consider in turn.

Program (a) is untyped. By default, our programming language is typed, so we indicate untyped code by surrounding it with ceiling brackets. Untyped code is really uni-typed (a slogan due to Robert Harper); it is a special case of typed code where every term has type `Dyn`. Here the term evaluates to $\lceil 4 \rceil : \texttt{Dyn}$.

(a) $\lceil \texttt{let}\ x = 2$
$\quad \texttt{let}\ f = \lambda y.\ y + 1$
$\quad \texttt{let}\ h = \lambda g.\ g\ (g\ x)$
$\quad \texttt{in}\ h\ f \rceil$

(b) $\texttt{let}\ x = 2$
$\quad \texttt{let}\ f = \langle \texttt{Int} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^p\ \lceil \lambda y.\ y + 1 \rceil$
$\quad \texttt{let}\ h = \lambda g : \texttt{Int} \to \texttt{Int}.\ g\ (g\ x)$
$\quad \texttt{in}\ h\ f$

(c) $\texttt{let}\ x = \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p\ 2$
$\quad \texttt{let}\ f = \langle \texttt{Nat} \to \texttt{Nat} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^q\ (\lambda y : \texttt{Int}.\ y + 1)$
$\quad \texttt{let}\ h = \lambda g : \texttt{Nat} \to \texttt{Nat}.\ g\ (g\ x)$
$\quad \texttt{in}\ \ h\ f$

(d) $\texttt{let}\ x = \lceil \texttt{true} \rceil$
$\quad \texttt{let}\ f = \lambda y : \texttt{Int}.\ y + 1$
$\quad \texttt{let}\ h = \langle (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^p\ \lceil \lambda g.\ g\ (g\ x) \rceil$
$\quad \texttt{in}\ h\ f$

(e) $\texttt{let}\ x = \lceil \texttt{true} \rceil$
$\quad \texttt{let}\ f = \langle \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p\ (\lambda y : \texttt{Int}.\ y + 1)$
$\quad \texttt{let}\ h = \lceil \lambda g.\ g\ (g\ x) \rceil$
$\quad \texttt{in}\ \lceil h\ f \rceil$

(f) $\texttt{let}\ x = \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p\ 3$
$\quad \texttt{let}\ f = \langle \texttt{Nat} \to \texttt{Nat} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^q\ (\lambda y : \texttt{Int}.\ y - 2)$
$\quad \texttt{let}\ h = \lceil \lambda g.\ g\ (g\ x) \rceil$
$\quad \texttt{in}\ \lceil h\ f \rceil$

Fig. 2. Example programs

As a matter of software engineering, when we add types to our code we may not wish to do so all at once. Progam (b) contains typed and untyped parts, fit together by casting the untyped code to a suitable type. and the term evaluates to $4 : \texttt{Int}$. Gradual evolution is overkill for such a short piece of code, but in real systems it plays an important role [17, 18]. Here $\lceil \lambda y.\ y + 1 \rceil$ has type $\texttt{Dyn}$, and the cast converts it to type $\texttt{Int} \to \texttt{Int}$.

In general, a cast from source type S to target type T is written $\langle T \Leftarrow S \rangle^p\ s$, where subterm s has type S and the whole term has type T, and p is a *blame label*. We assume an involutive operation of negation on blame labels: if p is a blame label then $\bar{p}$ is its negation, and $\overline{\bar{p}}$ is the same as p. Consider a cast with blame label p: blame is allocated to p when the *term contained* in the cast fails to satisfy the contract associated with the cast, while blame is allocated to $\bar{p}$ when the *context containing* the cast fails to satisfy the contract.

Our notation is chosen for clarity rather than compactness. Writing the source type is redundant, but convenient for a core calculus. Our notation is based on that of Gronski and Flanagan [7].

2.2 Contracts and Subset Types

Findler and Felleisen [4] introduced higher-order contracts, and Flanagan [5] and Ou *et al.* [13] observed that contracts can be incorporated into a type system as a form

of *subset* (or *refinement*) type. An example is $\{x : \texttt{Int} \mid x \geq 0\}$, the type of all integers greater than zero, which we will write `Nat`. A cast from `Int` to `Nat` performs a dynamic test, checking that the integer is indeed greater than or equal to zero. Just as we can start with an untyped program and add types, we can start with a typed program and add subset types. Program (c) is a version of the previous program with subset types added.

Unlike hybrid or dependent types, the blame calculus does not require subsumption. As a technical nicety, this allows us to design a type system which (like gradual types) satisfies *unicity*: every well-typed term has exactly one type. In order to achieve unicity, we must add new value forms corresponding to the result of casting to a subset type. Thus, the value of Program (c) is not $4 : \texttt{Int}$ but $4_{\texttt{Nat}} : \texttt{Nat}$.

2.3 The Blame Game

The above examples execute with no errors, but in general we may not be so lucky. Casts perform dynamic tests at run-time that fail if a value cannot be coerced to the given type. A cast to a subset type reduces to a dynamic test of the condition on the type. Recall that `Nat` denotes $\{x : \texttt{Int} \mid x \geq 0\}$. Here is a successful test:

$$\langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p \, 4 \quad \longrightarrow \quad 4 \geq 0 \rhd^p 4_{\texttt{Nat}} \quad \longrightarrow \quad \texttt{true} \rhd^p 4_{\texttt{Nat}} \quad \longrightarrow \quad 4_{\texttt{Nat}}$$

And here is a failed test:

$$\langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p \, -4 \quad \longrightarrow \quad -4 \geq 0 \rhd^p -4_{\texttt{Nat}} \quad \longrightarrow \quad \texttt{false} \rhd^p -4_{\texttt{Nat}} \longrightarrow \Uparrow p$$

The middle steps show a new term form that performs a dynamic test, of the form $s \rhd^p v_{\{x:B|t\}}$. If s evaluates to true, the value of subset type is returned; if s evaluates to false, blame is allocated to p, written $\Uparrow p$.

Given an arbitrary term that takes integers to integers, it is not decidable whether it also takes naturals to naturals. Therefore, when casting a function type the test is deferred until the function is applied. This is the essence of higher-order contracts.

Here is an example of casting a function and applying the result.

$$\begin{array}{l} (\langle \texttt{Nat} \to \texttt{Nat} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p \, (\lambda y : \texttt{Int}.\ y + 1)) \, 2_{\texttt{Nat}} \longrightarrow \\ \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p ((\lambda y : \texttt{Int}.\ y + 1) \, (\langle \texttt{Int} \Leftarrow \texttt{Nat} \rangle^{\bar{p}} \, 2_{\texttt{Nat}}) \longrightarrow \\ \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p \, ((\lambda y : \texttt{Int}.\ y + 1) \, 2) \longrightarrow \\ \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p \, 3 \longrightarrow 3_{\texttt{Nat}} \end{array}$$

The cast on the function breaks into two casts, each in opposite directions: the cast on the result takes the range of the *source* to the range of the *target*, while the cast on the argument takes the domain of the *target* to the domain of the *source*. Preserving order for the range while reversing order for the domain is analogous to the standard rule for function subtyping, which is covariant in the range and contravariant in the domain.

Observe that the blame label on the reversed cast has been negated, because if that cast fails it is the fault of the context, which supplies the argument to the function. Conversely, the blame label is not negated on the result cast, because if that cast fails it is the fault of the function itself.

The above cast took a function with range and domain `Int` to a function with more precise range and domain `Nat`. Now consider a cast to a function with less precise range and domain `Dyn`.

$$(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p\ (\lambda y : \texttt{Int}.\ y + 1))\ \lceil 2 \rceil \longrightarrow$$
$$\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^p((\lambda y : \texttt{Int}.\ y + 1)\ (\langle \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{\bar{p}}\ \lceil 2 \rceil)) \longrightarrow$$
$$\langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^p((\lambda y : \texttt{Int}.\ y + 1)\ 2) \longrightarrow \langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^p\ 3 \longrightarrow \lceil 3 \rceil$$

Again, a cast on the function breaks into two casts, each in opposite directions.

If we consider a well-typed term of the form

$$(\langle \texttt{Nat} \to \texttt{Nat} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p\ f)\ x$$

we can see that negative blame *never* adheres to this cast, because the type checker guarantees that x has type `Nat`, and the cast from `Nat` to `Int` always succeeds. However positive blame may adhere, for instance if f is $\lambda y : \texttt{Int}.\ y - 2$ and x is 1.

Conversely, if we consider a well-typed term of the form

$$(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p\ f)\ x$$

we can see that positive blame *never* adheres to this cast, because the types guarantee that f returns an `Int`, and the cast from `Int` to `Dyn` always succeeds. However negative blame may adhere, for instance if f is $\lambda y : \texttt{Int}.\ y + 1$ and x is $\lceil \texttt{true} \rceil$.

The key result of this paper is to show that casting from a more precise type to a less precise type cannot give rise to positive blame (but may give rise to negative); and that casting for a less precise type to a more precise type cannot give rise to negative blame (but may give rise to positive). Here are the two examples considered above, with the more precise type on the left, and the less precise type on the right.

$$\texttt{Nat} \to \texttt{Nat} <:_n \texttt{Int} \to \texttt{Int} \qquad\qquad \texttt{Int} \to \texttt{Int} <:_n \texttt{Dyn} \to \texttt{Dyn}$$

We call this *naive* subtyping (hence the subscript n) because it is covariant in both the domain and the range of function types, in contrast to traditional subtyping, which is contravariant in the domain and covariant in the range. We formally define both subtyping and naive subtyping in Section 3.3.

2.4 Well-Typed Programs Can't Be Blamed

Consider a program that mixes typed and untyped code; it will contain two sorts of casts. One sort gives types to untyped code. Such casts make types more precise, and so cannot give rise to negative blame. For instance, Program (d) in Figure 2 fails blaming p. Because the blame is positive, the fault lies with the untyped code inside the cast.

The other sort takes typed code and makes it untyped. Such a cast makes types less precise, and so cannot give rise to positive blame. For instance, Program (e) fails blaming $\bar{p}$. Because the blame is negative, the fault lies with the code outside the cast.

Both times the fault lies with the untyped code! This is of course what we would expect, since typed code should contain no type errors. Understanding positive and negative blame, and knowing when each can arise, is the key to giving a simple proof of this expected fact.

Syntax

variables	x, y	blame labels	p, q
base types	B	$::= \texttt{Bool} \mid \texttt{Int} \mid \cdots$	
constants	c	$::= \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid \cdots \mid + \mid - \mid \geq \cdots$	
types	S, T	$::= \texttt{Dyn} \mid B \mid S \to T \mid \{x : B \mid t\}$	
terms	s, t, u	$::= x \mid c \mid \lambda x : S.\ t \mid t\ s \mid \langle T \Leftarrow S \rangle^p\ s$	

Compile-time typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{T = \mathrm{ty}(c)}{\Gamma \vdash c : T}$$

$$\frac{\Gamma,\, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.\ t : S \to T} \qquad \frac{\Gamma \vdash t : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T} \qquad \frac{\Gamma \vdash s : S \qquad S \sim T}{\Gamma \vdash \langle T \Leftarrow S \rangle^p\ s : T}$$

Compatibility $\boxed{S \sim T}$

$$B \sim B \qquad \texttt{Dyn} \sim T \qquad S \sim \texttt{Dyn}$$

$$\frac{S \sim S' \qquad T \sim T'}{S \to T \sim S' \to T'} \qquad \frac{B \sim T}{\{x : B \mid s\} \sim T} \qquad \frac{S \sim B}{S \sim \{y : B \mid t\}}$$

Fig. 3. Compile-time types

The same analysis generalizes to code containing subset types. For instance, Program (f) fails blaming q. In this case, both casts make the types more precise, so cannot give rise to negative blame. Because the blame is positive, the fault lies with the less refined code inside the cast.

3 Types, Reduction, Subtyping

Compile-time type rules of our system are presented in Figure 3, run-time type rules and reduction rules in Figure 4, and rules for subtyping in Figure 5. We discuss each of these in turn in the following three subsections.

3.1 Types and Terms

Figure 3 presents the syntax of types and terms and the compile-time type rules. The language is explicitly and statically typed. (See Section 3.4 for embedding untyped terms.)

We let S, T range over types, and s, t range over terms. A type is either a base type B, the dynamic type $\texttt{Dyn}$, a function type $S \to T$, or a subset type $\{x : B \mid t\}$. A term is either a variable x, a constant c, a lambda expression $\lambda x : S.\ t$, an application $s\ t$, or a cast expression $\langle T \Leftarrow S \rangle^p\ s$. We write $\texttt{let}\ x = s\ \texttt{in}\ t$ as an abbreviation for $(\lambda x : S.\ t)\ s$ where s has type S.

We assume a denumerable set of constants. Every constant c is assigned a unique type $\mathrm{ty}(c)$. We assume $\texttt{Bool}$ is a base type with $\texttt{true}$ and $\texttt{false}$ as constants of type $\texttt{Bool}$; and that $\texttt{Int}$ is a base type with 0, 1, and so on, as constants of type $\texttt{Int}$, and $+$ and $-$ as constants of type $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$, and $\geq$ as a contant of type $\texttt{Int} \to \texttt{Int} \to \texttt{Bool}$, and possibly other constants. Constants must have base type or function type; this guarantees that every value of type $\texttt{Dyn}$ has the form $\texttt{Dyn}_G(v)$ and that every value of subset type has the form $v_{\{x:B|t\}}$. Constants of function type must not raise blame when evaluated; this guarantees that only casts can raise blame.

Syntax

ground types	G	$::= B \mid \texttt{Dyn} \to \texttt{Dyn}$
terms	s, t, u	$::= \cdots \mid \texttt{Dyn}_G(v) \mid v_{\{x:B\mid t\}} \mid s \rhd^p v_{\{x:B\mid t\}}$
values	v, w	$::= x \mid c \mid \lambda x : S.\ t \mid \langle S' \to T' \Leftarrow S \to T\rangle^p\ v \mid \texttt{Dyn}_G(v) \mid v_{\{x:B\mid t\}}$
results	r	$::= t \mid \Uparrow p$
eval contexts	E	$::= [\,] \mid E\ s \mid v\ E \mid \langle T \Leftarrow S\rangle^p\ E \mid E \rhd^p v_{\{x:B\mid t\}}$

Run-time typing

$$\frac{\Gamma \vdash s : \texttt{Bool} \qquad \Gamma \vdash v : B \qquad t[x := v] \longrightarrow^* s}{\Gamma \vdash s \rhd^p v_{\{x:B\mid t\}} : \{x : B \mid t\}} \qquad \boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash v : G}{\Gamma \vdash \texttt{Dyn}_G(v) : \texttt{Dyn}} \qquad \frac{\Gamma \vdash v : B \qquad t[x := v] \longrightarrow^* \texttt{true}}{\Gamma \vdash v_{\{x:B\mid t\}} : \{x : B \mid t\}}$$

Reductions $\boxed{s \longrightarrow r}$

$$E[c\ v] \longrightarrow E[[\![c]\!](v)] \tag{1}$$
$$E[(\lambda x : S.\ t)\ v] \longrightarrow E[t[x := v]] \tag{2}$$
$$E[\langle B \Leftarrow B\rangle^p\ v] \longrightarrow E[v] \tag{3}$$
$$E[(\langle S' \to T' \Leftarrow S \to T\rangle^p\ v)\ w] \longrightarrow E[\langle T' \Leftarrow T\rangle^p\ (v\ (\langle S \Leftarrow S'\rangle^{\bar{p}}\ w))] \tag{4}$$
$$E[\langle \texttt{Dyn} \Leftarrow \texttt{Dyn}\rangle^p\ v] \longrightarrow E[v] \tag{5}$$
$$E[\langle \texttt{Dyn} \Leftarrow B\rangle^p\ v] \longrightarrow E[\texttt{Dyn}_B(v)] \tag{6}$$
$$E[\langle \texttt{Dyn} \Leftarrow S \to T\rangle^p\ v] \longrightarrow E[\texttt{Dyn}_{\texttt{Dyn}\to\texttt{Dyn}}(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow S \to T\rangle^p\ v)] \tag{7}$$
$$E[\langle T \Leftarrow \texttt{Dyn}\rangle^p\ \texttt{Dyn}_G(v)] \longrightarrow E[\langle T \Leftarrow G\rangle^p\ v], \quad \text{if } G \sim T \tag{8}$$
$$E[\langle T \Leftarrow \texttt{Dyn}\rangle^p\ \texttt{Dyn}_G(v)] \longrightarrow \Uparrow p, \quad \text{if } G \not\sim T \tag{9}$$
$$E[\langle \{x : B \mid t\} \Leftarrow S\rangle^p\ v] \longrightarrow E[\texttt{let}\ x = \langle B \Leftarrow S\rangle^p\ v\ \texttt{in}\ t \rhd^p x_{\{x:B\mid t\}}] \tag{10}$$
$$E[\texttt{true} \rhd^p v_{\{x:B\mid t\}}] \longrightarrow E[v_{\{x:B\mid t\}}] \tag{11}$$
$$E[\texttt{false} \rhd^p v_{\{x:B\mid t\}}] \longrightarrow \Uparrow p \tag{12}$$
$$E[\langle T \Leftarrow \{x : B \mid s\}\rangle^p\ v_{\{x:B\mid s\}}] \longrightarrow E[\langle T \Leftarrow B\rangle^p\ v] \tag{13}$$

Fig. 4. Run-time types and reduction

The type system is explained in terms of three judgements, which are presented in Figure 3. We write $\Gamma \vdash t : T$ if term t has type T in environment Γ, and we write $S \sim T$ if type S is compatible with type T. We let Γ range over type environments, which are a list of variable-type pairs $x : T$.

A type is well-formed if for every subset type $\{x : B \mid t\}$ we have that t has type $\texttt{Bool}$ on the assumption that x has type B (no other free variables may appear in t). In what follows, we assume all types are well-formed. We call B the *domain* of the subset type $\{x : B \mid t\}$.

The type rules for variables, constants, lambda abstraction, and application are standard. The type rule for casts is straightforward: if term s has type S and type S is compatible with type T (defined below), then the term $\langle T \Leftarrow S\rangle^p\ s$ has type T.

We write $S \sim T$ for the *compatibility* relation, which holds if it may be sensible to cast type S to type T. A base type is compatible with itself, type $\texttt{Dyn}$ is compatible with any type, two function types are compatible if their domains and ranges are compatible, and a subset type is compatible with every type that is compatible with its domain.

Compatibility is reflexive and symmetric but not transitive. For example, $S \sim \texttt{Dyn}$ and $\texttt{Dyn} \sim T$ hold for any types S and T, but $S \sim T$ does not hold if one of S or T is a function type and the other is a base type. Requiring compatibility ensures that there are no obviously foolish casts, but does not rule out the possibility of two successive casts, one from S to $\texttt{Dyn}$ and the next from $\texttt{Dyn}$ to T.

Our cast rule is inspired by the similar rules found for gradual types and hybrid types. Gradual types introduce compatibility, but do not have subset types. Hybrid types include subset types, but do not bother with compatibility. Neither system uses both positive and negative blame labels, as we do here.

Hybrid types also have a subsumption rule: if s has type S, and S is a subtype of T, then s also has type T. This greatly increases the power of the type system. For instance, in hybrid types each constant is assigned the singleton type $c : \{x : B \mid c = x\}$; and by subtyping and subsumption it follows that each constant belongs to every subset type $\{x : B \mid t\}$ for which $t[x := c] \longrightarrow^* \texttt{true}$. However, the price paid for this is that type checking for hybrid types is undecidable, because the subtype relation is undecidable.

A pleasant consequence of omitting subsumption from the blame calculus is that each term has a unique type, and an even more pleasant consequence is that the type system for the source language is decidable.

Proposition 1. *(Unicity) If $\Gamma \vdash s : S$ and $\Gamma \vdash s : T$ then $S = T$.*

Proposition 2. *(Decidability) Given Γ and t, it is decidable whether there is a T such that $\Gamma \vdash t : T$ (using the compile-time type rules of Figure 3).*

Both propositions are easy inductions.

However, there are some less pleasant consequences. (The tiger is caged, not tamed!) Reduction may introduce terms that are not permitted in the source language, and we need additional semidecidable run-time rules to check the types of these terms. We explain the details of how this works below.

3.2 Reductions

Figure 4 defines additional term forms, values, evaluation contexts, additional run-time type rules, and reduction.

We let v, w range over values. A value is either a variable, a constant, a lambda term, a cast to a function type from another function type, an injection into dynamic from a ground type, or an injection into a subset type from its domain type. The first three of these are standard, and we explain the other three below.

We take a cast to a function type from another function type as a value for technical convenience. Other work [15, 5] makes the opposite choice, and reduce a cast to a function type from another function type to a lambda expression.

Values of dynamic type take the form $\texttt{Dyn}_G(v)$, where G is ground type, which is either a base type B or the function type $\texttt{Dyn} \to \texttt{Dyn}$, and v is a value of type G. For

example, the cast $\langle \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p\ (\lambda x : \texttt{Int}.\ x + 1)$ reduces to the value $\texttt{Dyn}_{\texttt{Dyn}\to\texttt{Dyn}}(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle^p\ (\lambda x : \texttt{Int}.\ x + 1))$. Note that the inner cast is a value, since it is to a function type from another function type.

Values of subset type take the form $v_{\{x:B|t\}}$ where v is a value of type B and $t[x := v] \longrightarrow^* \texttt{true}$. We also need an intermediate term to test the predicate associated with a cast to a subset type. This term has the form $s \rhd^p v_{\{x:B|t\}}$, where v is a value of type T, and s is a boolean term such that $t[x := v] \longrightarrow^* s$. If s reduces to $\texttt{true}$ the term reduces to $v_{\{x:B|t\}}$, and if s reduces to $\texttt{false}$ the term allocates blame to p.

(In contrast, Flanagan [5] has essentially the following rule.

$$\frac{t[x := v] \longrightarrow^* \texttt{true}}{\langle \{x : B \mid t\} \Leftarrow B \rangle^p\ v \longrightarrow v_{\{x:B|t\}}}$$

This formulation is unusual, in that a single reduction step in the conclusion depends on multiple steps in the hypothesis. The rule makes it awkward to formulate a traditional progress theorem, because if reduction of $t[x := v]$ proceeds forever, then evaluation gets stuck.)

We let E range over evaluation contexts, which are standard. The cast operation is strict, and reduces the term being cast to a value before the cast is performed, and the subset test is strict in its predicate.

We write $s \longrightarrow r$ to indicate that a single reduction step takes term s to result r, which is either a term t or the form $\Uparrow p$, which indicates allocation of blame to label p. We write $s \longrightarrow^* r$ for the reflexive and transitive closure of reduction.

There are three additional type rules for the three additional term forms. These are straightforward, save that the two rules for subset types involve reduction, and hence are semi-decidable. Hence, Proposition 2 (Decidability) holds only for the compile-time syntax type rules of Figure 3, and fails when these are extended with the run-time type rules of Figure 4. However, it is easy to check that Proposition 1 (Unicity), holds even when the compile-time type rules are extended with the run-time type rules.

The good news is that semi-decidability is not a show stopper. We introduce the semi-decidable type rules precisely in order to prove preservation and progress. Typing of the source language is decidable, and reduction is decidable. We never need to check whether a term satisfies the semi-decidable rules, since this is guaranteed by preservation and progress!

We now go through each of the reductions in turn. (1) Constants of function type are interpreted by a semantic function consistent with their type: if $\text{ty}(c) = S \to T$ and value v has type S, then $[\![c]\!](v)$ is a term of type T. For example, $\text{ty}(+) = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$, with $[\![+]\!](3) = +_3$, where $\text{ty}(+_3) = \texttt{Int} \to \texttt{Int}$ and $[\![+_3]\!](4) = 7$. (2) The rule for applying a lambda expression is standard. (3) A cast to a base type from itself is the identity. (4) A cast to a function type from another function type decomposes into separate casts on the argument and result. Note the reversal in the argument cast, and the corresponding negating of the blame label.

The next three rules concern casts to the dynamic type. (5) A cast to $\texttt{Dyn}$ from itself is the identity. (6) A cast to $\texttt{Dyn}$ from a base type is a value. (7) A cast to $\texttt{Dyn}$ from a function type $S \to T$ decomposes into a cast to $\texttt{Dyn}$ from the ground type $\texttt{Dyn} \to \texttt{Dyn}$, and a cast to $\texttt{Dyn} \to \texttt{Dyn}$ from $S \to T$.

Entailment $\boxed{x : T \Leftarrow S \models t}$

$$\frac{S \sim T \qquad \text{for all } v \text{ and } w, \text{ if } \vdash v : S \text{ and } \langle T \Leftarrow S \rangle^p\ v \longrightarrow^* w \text{ then } t[x := w] \longrightarrow^* \texttt{true}}{x : T \Leftarrow S \models t}$$

Subtype $\boxed{S <: T}$

$$B <: B \qquad \texttt{Dyn} <: \texttt{Dyn}$$

$$\frac{S' <: S \qquad T <: T'}{S \to T <: S' \to T'} \qquad \frac{B <: T}{\{x : B \mid s\} <: T} \qquad \frac{S <: B \qquad x : B \Leftarrow S \models t}{S <: \{x : B \mid t\}} \qquad \frac{S <: G}{S <: \texttt{Dyn}}$$

Positive subtype $\boxed{S <:^+ T}$

$$B <:^+ B \qquad S <:^+ \texttt{Dyn}$$

$$\frac{S' <:^- S \qquad T <:^+ T'}{S \to T <:^+ S' \to T'} \qquad \frac{B <:^+ T}{\{x : B \mid s\} <:^+ T} \qquad \frac{S <:^+ B \qquad x : B \Leftarrow S \models t}{S <:^+ \{x : B \mid t\}}$$

Negative subtype $\boxed{S <:^- T}$

$$B <:^- B \qquad \texttt{Dyn} <:^- T$$

$$\frac{S' <:^+ S \qquad T <:^- T'}{S \to T <:^- S' \to T'} \qquad \frac{B <:^- T}{\{x : B \mid s\} <:^- T} \qquad \frac{S <:^- B}{S <:^- \{x : B \mid t\}} \qquad \frac{S <:^- G}{S <:^- T}$$

Naive subtype $\boxed{S <:_n T}$

$$B <:_n B \qquad S <:_n \texttt{Dyn}$$

$$\frac{S <:_n S' \qquad T <:_n T'}{S \to T <:_n S' \to T'} \qquad \frac{B <:_n T}{\{x : B \mid s\} <:_n T} \qquad \frac{S <:_n B \qquad x : B \Leftarrow S \models t}{S <:_n \{x : B \mid t\}}$$

Fig. 5. Subtypes

The next two rules concern casts from the dynamic type. (8) A cast to type T from the value $\texttt{Dyn}_G(v)$ of dynamic type collapses to a cast to type T directly from type G if the types T and G are compatible. (9) Otherwise, such a cast fails.

The next three rules concern casts to subset type. (10) A cast to subset type with domain B from type S decomposes into a cast to B from S, followed by a test that the value satisfies the predicate. (11) If the predicate evaluates to true the test reduces to the subset type. (12) Otherwise the test fails.

The last rule concerns casts from a subset type. (13) Consider a cast to type T from a subset type. Recall that values of subset type have the form $v_{\{x:B|s\}}$, where v has type B. The cast collapses to a cast directly to T from B. Note that B and T must be compatible, since a subset type is only compatible with a type that is compatible with its domain.

3.3 Subtyping

We do not need subtyping to assign types to terms, but we will use subtyping to characterise when a cast cannot give rise to blame. Figure 5 presents entailment and four subtyping judgements—ordinary, positive, negative, and naive.

Entailment is written

$$x : T \Leftarrow S \models t$$

and holds if for all values v of type S and w of type T such that $\langle T \Leftarrow S \rangle^p\ v \longrightarrow^* w$ we have that $t[x := w] \longrightarrow^* \texttt{true}$.

We write $S <: T$ if S is a subtype of T. Function subtyping is contravariant in the domain and covariant in the range. A subset type is a subtype of its domain, and a type is a subtype of a subset type if membership in the type entails satisfaction of the subset type's predicate. Every subtype of a ground type is a subtype of `Dyn`, since casts from subtypes of a ground type to `Dyn` cannot allocate blame.

For example, say that we define $\texttt{Pos} = \{x : \texttt{Int} \mid x > 0\}$ and $\texttt{Nat} = \{x : \texttt{Int} \mid x \geq 0\}$. Then $x : \texttt{Int} \Leftarrow \texttt{Pos} \models x \geq 0$, and so $\texttt{Pos} <: \texttt{Nat}$ by the sixth rule. For another example, $\texttt{Int} <: \texttt{Int}$ by the first rule, so $\texttt{Pos} <: \texttt{Int}$ by the fifth rule, so $\texttt{Pos} <: \texttt{Dyn}$ by the third rule.

Entailment, and hence subtyping, are undecidable. This is not a hindrance, since our type system does not depend on subtyping. Rather it is an advantage, since it means we can show more types are in the subtype relation, making our results more powerful.

Our rules for subtyping are similar to those found in earlier work [5, 8, 13]. However, they take every type to be a subtype of `Dyn`. In contrast, we only take S to be a subtype of T if a cast from S to T can never receive any blame, and therefore the only subtypes of `Dyn` are `Dyn` itself and subtypes of ground types. It is not appropriate to take function types (other that $\texttt{Dyn} \to \texttt{Dyn}$) as subtypes of `Dyn`, because a cast to `Dyn` from a function type may receive negative blame. The issues are similar to the treatment of the contract `Any` [3].

In order to characterize when positive and negative blame cannot occur, we factor subtyping into two subsidiary relations, positive subtyping, written $S <:^+ T$ and negative subtyping, written $S <:^- T$. The two judgements are defined in terms of each other, and track the swapping of positive and negative blame labels that occurs with function types, with the contravariant position in the function typing rule reversing the roles. We have $S <:^+ \texttt{Dyn}$ and $\texttt{Dyn} <:^- T$ for every type S and T, since casting to `Dyn` can never give rise to positive blame, and casting from `Dyn` can never give rise to negative blame. We only check entailment between subtypes for positive subtyping, since failure of a subset predicate gives rise to positive blame. Finally, on the negative side, if $S <:^- G$, then $S <:^- T$, since no cast from a subtype of a ground type to any other type can allocate negative blame.

Proposition 3. *(Subtyping is transitive and reflexive) If $S <: S'$ and $S' <: S''$ then $S <: S''$, for all S, S', S'', and $S <: S$, for all S. Similarly for $<:^+$, $<:^-$, and $<:_n$.*

Proposition 4. *(Subtyping and compatibility) If $S <: T$ then $S \sim T$. Similarly for $<:^+$ and $<:_n$, but not $<:^-$.*

The main results concerning positive and negative subtyping are given in Section 4. We show that $S <: T$ if and only if $S <:^+ T$ and $S <:^- T$. We also show that if $S <:^+ T$ then a cast from S to T cannot receive positive blame, and that if $S <:^- T$ then a cast from S to T cannot receive negative blame.

We also define a naive subtyping judgement, $S <:_n T$, which corresponds to our informal notion of type S being more precise than type T, and is covariant for both

Syntax

$$\text{untyped terms} \quad M, N ::= x \mid k \mid \lambda x.\ N \mid M\ N \mid \lfloor t \rfloor$$

Well-formed terms $\boxed{\Gamma \vdash M \text{ wf}}$

$$\frac{(x : \texttt{Dyn}) \in \Gamma}{\Gamma \vdash x \text{ wf}} \qquad \frac{\Gamma,\, x : \texttt{Dyn} \vdash N \text{ wf}}{\Gamma \vdash (\lambda x.\ N) \text{ wf}} \qquad \frac{\Gamma \vdash M \text{ wf} \qquad \Gamma \vdash N \text{ wf}}{\Gamma \vdash (M\ N) \text{ wf}} \qquad \frac{\Gamma \vdash t : \texttt{Dyn}}{\Gamma \vdash \lfloor t \rfloor \text{ wf}}$$

Embedding $\boxed{\lceil M \rceil}$

$$\begin{array}{ll} \lceil x \rceil & = x \\ \lceil c \rceil & = \langle \texttt{Dyn} \Leftarrow \text{ty}(c) \rangle\ c \\ \lceil \lambda x.\ N \rceil & = \langle \texttt{Dyn} \Leftarrow \texttt{Dyn} \to \texttt{Dyn} \rangle\ (\lambda x : \texttt{Dyn}.\ \lceil N \rceil) \\ \lceil M\ N \rceil & = (\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Dyn} \rangle\ \lceil M \rceil)\ \lceil N \rceil \\ \lceil \lfloor t \rfloor \rceil & = t \end{array}$$

Fig. 6. Untyped lambda calculus

the domain and range of functions. In Section 4, we show that $S <:_n T$ if and only if $S <:^+ T$ and $T <:^- S$. (Note the reversal! In the similar statement for ordinary subtyping, we wrote $S <:^- T$, where here we write $T <:^- S$.)

Here are some examples:

$$\begin{array}{ll} \texttt{Int} \to \texttt{Nat} <: \ \texttt{Nat} \to \texttt{Int} & \texttt{Nat} \to \texttt{Nat} <:_n \texttt{Int} \to \texttt{Int} \\ \texttt{Int} \to \texttt{Nat} <:^+ \texttt{Nat} \to \texttt{Int} & \texttt{Nat} \to \texttt{Nat} <:^+ \texttt{Int} \to \texttt{Int} \\ \texttt{Int} \to \texttt{Nat} <:^- \texttt{Nat} \to \texttt{Int} & \texttt{Int} \to \texttt{Int} <:^- \texttt{Nat} \to \texttt{Nat} \end{array}$$

The left-hand side line shows that ordinary subtyping is contravariant in the domain and covariant in the range, while the right-hand side shows that naive subtyping is covariant in both. In both cases, the first line is equivalent to the second and third.

3.4 Typed and Untyped Lambda Calculus

We introduce a separate grammar for untyped terms, and show how to embed untyped terms into typed terms (and vice versa). The relevant definitions are in Figure 6.

Let M, N range over untyped terms. The term form $\lfloor t \rfloor$ lets us embed typed terms into untyped terms; it is well-formed only if the typed term t has type $\texttt{Dyn}$. Below we define a mapping $\lceil M \rceil$, that lets us embed untyped terms into typed terms.

An untyped term is well-formed if every variable appearing free in it has type $\texttt{Dyn}$, and if every typed subterm has type $\texttt{Dyn}$. We write $\Gamma \vdash M$ wf to indicate that M is well-formed.

A simple mapping takes untyped terms into typed terms. An untyped term M is well-formed if and only if the corresponding typed term $\lceil M \rceil$ is well-typed with type $\texttt{Dyn}$.

Lemma 1. *We have* $\Gamma \vdash M$ wf *if and only if* $\Gamma \vdash \lceil M \rceil : \texttt{Dyn}$.

It is straightforward to define reduction for untyped terms, and show that the embedding preserves and reflects reductions.

$$\frac{S <:^{+} T \qquad s \text{ safe for } p}{\langle T \Leftarrow S\rangle^{p} s \text{ safe for } p} \qquad \frac{S <:^{-} T \qquad s \text{ safe for } p}{\langle T \Leftarrow S\rangle^{\bar{p}} s \text{ safe for } p} \qquad \frac{p \neq q \qquad \bar{p} \neq q \qquad s \text{ safe for } p}{\langle T \Leftarrow S\rangle^{q} s \text{ safe for } p}$$

$$\frac{v \text{ safe for } p}{\mathtt{Dyn}_{G}(v) \text{ safe for } p} \qquad \frac{s \longrightarrow^{*} \mathtt{true}}{s \rhd^{p} v_{\{x:B|t\}} \text{ safe for } p} \qquad \frac{q \neq p \qquad s \text{ safe for } p}{s \rhd^{q} v_{\{x:B|t\}} \text{ safe for } p}$$

$$\frac{}{x \text{ safe for } p} \qquad \frac{}{c \text{ safe for } p} \qquad \frac{t \text{ safe for } p}{\lambda x : S.\ t \text{ safe for } p} \qquad \frac{t \text{ safe for } p \qquad s \text{ safe for } p}{t\ s \text{ safe for } p}$$

Fig. 7. Safe terms

3.5 Type Safety

We have usual substitution and canonical forms lemmas, and preservation and progress results.

Lemma 2. *(Substitution) If $\Gamma \vdash v : S$ and $\Gamma, x : S \vdash t : T$, then $\Gamma \vdash t[x := v] : T$.*

Lemma 3. *(Canonical forms) Let v be a value that is well-typed in the empty context. One of three cases applies.*

- *If $\vdash v : S \to T$ then either*
  - *$v = \lambda x : S.\ t$, with $x : S \vdash t : T$, or*
  - *$v = c$, with $\mathrm{ty}(c) = S \to T$, or*
  - *$v = \langle S \to T \Leftarrow S' \to T'\rangle^{p}\ v'$ with $\vdash v' : S' \to T'$.*
- *If $\vdash v : \{x : B \mid t\}$ then $v = v'_{\{x:B|t\}}$ with $\vdash v' : B$ and $t[x := v'] \longrightarrow^{*} \mathtt{true}$.*
- *If $\vdash v : \mathtt{Dyn}$ then $v = \mathtt{Dyn}_{G}(v')$ with $\vdash v' : G$.*

Proposition 5. *(Preservation) If $\Gamma \vdash s : T$ and $s \longrightarrow t$ then $\Gamma \vdash t : T$.*

Proposition 6. *(Progress) If $\vdash s : T$ then either*

- *s is a value, or*
- *$s \longrightarrow t$ for some result t, or*
- *$s \longrightarrow \Uparrow p$ for some blame label p.*

In this case, preservation and progress do not guarantee a great deal, since they do not rule out blame as a result. However, Section 4 gives results that let us identify circumstances where certain kinds of blame cannot arise.

4 The Blame Theorem

Subtyping factors into positive and negative subtyping, and naive subtyping also factors into positive and negative subtyping, this time with the direction of negative subtyping reversed.

Proposition 7. *(Factoring subtyping) We have $S <: T$ if and only if $S <:^{+} T$ and $S <:^{-} T$.*

Proposition 8. *(Factoring naive subtyping) We have $S <:_n T$ if and only if $S <:^+ T$ and $T <:^- S$.*

The following is the central result of this paper and depends on the definition of the safe for relation. A term t is safe for a blame label p if all of the casts that have the label p are positive subtypes, all of the casts that have the label $\neg p$, are negative subtypes, and all of the predicate tests with the label p succeed. The precise definition is given in Figure 7.

Proposition 9. *(Preservation of safe terms) For any well-typed term t and blame label p, if t* safe for *p and $t \longrightarrow t'$ then t'* safe for *p.*

Proposition 10. *(Progress of safe terms) For any well-typed term t and blame label p, if t* safe for *p then $t \not\longrightarrow \Uparrow p$.*

Corollary 1. *(Well-typed programs can't be blamed) Let t be a well-typed term with a subterm $\langle T \Leftarrow S \rangle^p$ s containing the only occurrences of p in t.*

- *If $S <:^+ T$ then $t \not\longrightarrow^* \Uparrow p$.*
- *If $S <:^- T$ then $t \not\longrightarrow^* \Uparrow \bar{p}$.*
- *If $S <: T$ then $t \not\longrightarrow^* \Uparrow p$ and $t \not\longrightarrow^* \Uparrow \bar{p}$*

In particular, since $S <:^+ \texttt{Dyn}$, any failure of a cast from a well-typed term to a dynamically-typed context must be blamed on the dynamically-typed context. And since $\texttt{Dyn} <:^- T$, any failure of a cast from a dynamically-typed term to a well-typed context must be blamed on the dynamically-typed term.

Further, consider a cast from a more precise type to a less precise type, which we can capture using naive subtyping. Since $S <:_n T$ implies $S <:^+ T$, any failure of a cast from a more-precisely-typed term to a less-precisely-typed context must be blamed on the less-precisely-typed context. And since $T <:_n S$ implies $S <:^- T$, any failure of a cast from a less-precisely-typed term to a more-precisely-typed context must be blamed on the less-precisely-typed term.

5 Related Work

Integrating static and dynamic typing is not new, and previous work includes type Dynamic [1], soft types [21], and partial types [16]. Contracts for dynamic testing of specifications were popularized by the language Eiffel [12]. Findler and Felleisen [4] introduced the use of higher-order contracts with blame in functional programming.

Henglein [9] lays much of the theoretical ground work for combining typed and untyped program fragments in a single program. Our work's principal technical debt concerns canonical coercions and the results surrounding them which justify our writing of casts as just a pair of types, instead of a pair of types combined with an explicit coercion (as Henglein does). Due to a coincidence of terminology, it is natural to compare Henglein's positive and negative coercions with our positive and negative subtyping relations, but they are essentially unrelated. Henglein's positive and negative coercions simply characterize naive subtyping [9, Proposition 23].

Siek and Taha [15] introduced gradual types, inspired by Gray et al [6]. Our results augment theirs, since we show how the blame for a failed cast always lies with the less-typed portion of the code. Siek, Garcia, and Taha [14] compare various approaches to subtyping for gradual types, including the one considered in this paper.

Flanagan et al [5, 8] introduced hybrid types and a new programming language, Sage. Ou et al [13] present a closely-related language with dynamically-checked dependent types. These support dependent function types, while our work here is restricted to ordinary function types.

Acknowledgements. This paper benefited enormously from conversations with John Hughes. Thanks to Samuel Bronson, Matthias Felleisen, Cormac Flanagan, Oleg Kiselyov, Jeremy Siek, and anonymous referees of earlier drafts for their comments on the paper. A special thanks to Michael Greenberg, Nate Foster, and Benjamin Pierce for discovering a technical flaw in an earlier version.

References

[1] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Trans. Prog. Lang. Syst. 13(2), 237–268 (1991)

[2] Bracha, G.: Pluggable type systems. In: OOPSLA 2004 Workshop on Revival of Dynamic Languages (October 2004)

[3] Findler, R., Blume, M.: Contracts as pairs of projections. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 226–241. Springer, Heidelberg (2006)

[4] Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ACM International Conference on Functional Programming (ICFP) (October 2002)

[5] Flanagan, C.: Hybrid type checking. In: ACM Symposium on Principles of Programming Languages (POPL) (Janurary 2006)

[6] Gray, K.E., Findler, R.B., Flatt, M.: Fine-grained interoperability through contracts and mirrors. In: ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), pp. 231–246 (2005)

[7] Gronski, J., Flanagan, C.: Unifying hybrid types and contracts. In: Trends in Functional Programming (TFP) (April 2007)

[8] Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Workshop on Scheme and Functional Programming (September 2006)

[9] Henglein, F.: Dynamic typing: Syntax and proof theory. Sci. Comput. Programming 22(3), 197–230 (1994)

[10] Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. In: ACM Symposium on Principles of Programming Languages (POPL) (Janurary 2007)

[11] Meijer, E.: Static typing where possible, dynamic typing where needed. In: OOPSLA 2004 Workshop on Revival of Dynamic Languages (October 2004)

[12] Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs (1988)

[13] Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: IFIP International Conference on Theoretical Computer Science (August 2004)

[14] Siek, J., Garcia, R., Taha, W.: Exploring the design space of higher-order casts. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 17–31. Springer, Heidelberg (2009)

[15] Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Workshop on Scheme and Functional Programming (September 2006)

[16] Thatte, S.: Type inference with partial types. In: Lepistö, T., Salomaa, A. (eds.) ICALP 1988. LNCS, vol. 317. Springer, Heidelberg (1988)
[17] Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Dynamic Languages Symposium (DLS) (2006)
[18] Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: ACM Symposium on Principles of Programming Languages (POPL) (2008)
[19] Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Workshop on Scheme and Functional Programming (September 2007)
[20] Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. Technical Report TR-2009-01, University of Chicago (2009)
[21] Wright, A.K., Cartwright, R.: A practical soft typing system for Scheme. ACM Trans. Prog. Lang. Syst. 19(1) (1997)