Well-typed programs can't be blamed

Philip Wadler
University of Edinburgh

Robert Bruce Findler
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 subtyping, and show that these recombine to yield naive subtyping. Naive typing has previously appeared in type systems that are unsound, but we believe this is the first time naive subtyping has played a role in establishing type soundness.

1. Introduction

Much recent work has focussed on how to integrate dynamic and static typing, using the contracts of Findler and Felleisen (2002) to ensure that dynamically-typed code meets statically-typed invariants. Examples include the *gradual types* of Siek and Taha (2006) the *hybrid types* of Flanagan (2006), the *dynamic dependent types* of Ou et al. (2004), and the *multi-language* programming of Matthews and Findler (2007); we list many others under related work.

Both Meijer (2004) and Bracha (2004) argue in favor of mixing dynamic and static typing. Static and dynamic typing are both supported in Visual Basic, and there are plans to bring similar integration to Perl 6 and ECMAScript 4. The latter, more popularly known as JavaScript, benefits from the involvement of Herman and Flanagan (2007), researchers known for their work on gradual and hybrid types (Herman et al. 2007).

Here, we provide a uniform view of much 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 refinement types (as with hybrid types).

The technical content of this paper is to introduce notions of positive and negative subtyping, and prove a theorem that characterises when positive and negative blame can occur. A corollary of this theorem 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.

A novel aspect of our work is that it involves both ordinary subtyping (which for functions is contravariant in the domain and covariant in the range) and naive subtyping (which for function is covariant in both the domain and the range). Ordinary subtyping characterizes a cast that cannot fail, while naive subtyping characterizes which side of a cast is less-typed (and hence will be blamed if the cast fails). We show that ordinary subtyping decomposes into positive and negative subtyping, and that these recombine in a different way to yield naive subtyping. A striking analogy is a tangram, where a square decomposes into parts that recombine into a different shape (see Figure 1). Naive subtyping has previously appeared in type systems that are unsound, notably that of Eiffel (Meyer 1988), but we believe this is the first time naive subtyping has played a role in establishing type soundness.

Many readers will recognise that our title is the third in a series. "Well-typed programs can't go wrong" summarised a denotational approach to soundness introduced by Milner (1978). "Well-typed programs don't get stuck" refined this slogan, summarising an operational approach to soundness introduced by Wright and Felleisen (1994). A related slogan, "safety is preservation plus progress", is due to Harper (Pierce 2002, page 95). "Well-typed programs can't be blamed" describes an approach suited to systems that use contracts and blame, characterising interaction between more-typed and less-typed components of a program.

Unlike the source languages for gradual and hybrid types, the blame calculus has the advantage that the source language makes clear where static guarantees hold and where dynamic checks are enforced. We therefore suggest that what has been used as an intermediate language for gradual and hybrid types is itself useful as a source language. We also suggest, in contrast to some previous work, that hybrid types can be useful even in the absence of a theorem prover—one need not have a sophisticated type checker to benefit from sophisticated types!—and that one should *not* regard every type as a subtype of the dynamic type.

We make the following contributions:

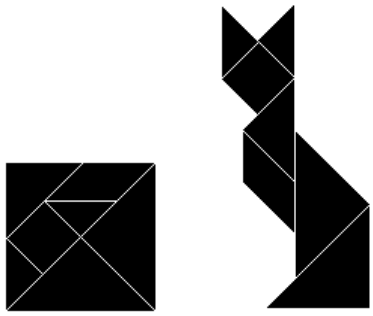

Figure 1. Tangrams as metaphor: Ordinary subtyping decomposes into components that recombine to yield naive subtyping.

- We introduce the blame calculus, showing that a language with explicit casts and no theorem prover is suited to many of the same purposes as gradual types and hybrid types (Section 2).
- We give a framework similar to that of the hybrid typing of Flanagan (2006) and the dynamic dependent typing of Ou et al. (2004), but with a decidable type system for the source language and satisfying unicity of type (Section 3).
- We factor the well-known notion of subtyping into new notions of positive and negative subtyping, and show that these recombine into naive subtyping. 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).
- We apply our theorem to sharpen published results for gradual types (Siek and Taha 2006) and hybrid types (Flanagan 2006), and to shed light on recently published results by Matthews and Findler (2007) and Gronski and Flanagan (2007) (Section 5).

Section 6 describes related work, and Section 7 concludes.[1]

2. The blame calculus

2.1 From Untyped to Typed

Consider the following untyped code.

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

By default, our programming language is typed, so we indicate untyped code by surrounding it with ceiling brackets. Untyped code is really uni-typed; it is a special case of typed code where every term has type `Dyn` (Harper 2007). The above term evaluates to $\lceil 4 \rceil : \texttt{Dyn}$.

As a matter of software engineering, when we add types to our code we may not wish to do so all at once. For instance, here is a version of the program containing typed and untyped parts; to fit them together, the untyped code is cast to a suitable type. Gradual evolution is overkill for such a short piece of code, but it should be clear how this would work in a larger system.

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

Here, $\lceil \lambda y.\ y + 1 \rceil$ has type `Dyn`, and the cast converts it to type $\texttt{Int} \to \texttt{Int}$. The above term evaluates to $4 : \texttt{Int}$.

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 $\bar{\bar{p}}$ is the same as p. Consider the failure of a cast with blame label p. Blame is allocated to p when it is the term contained the cast that fails to satisfy the contract associated with the cast, while blame is allocated to $\bar{p}$ when it is the context containing the cast that 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. Even writing the target can be cumbersome. The gradual type system of Siek and Taha (2006), the hybrid type system of Flanagan (2006), and the dynamic dependent types of Ou et al. (2004) use source languages where most or all casts are omitted, but inferred by a type-directed translation (and all three use quite similar translations). However, we prefer to have an indication in the source code of where casts are required. Our notation is based on that in Gronski and Flanagan (2007).

2.2 Contracts and refinement types

Findler and Felleisen (2002) introduced higher-order contracts, and Flanagan (2006) observed that contracts can be incorporated into a type system as a form of refinement type.

An example of a refinement type 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 refinement types. Here is a version of the previous program with refinement types added.

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

The hybrid type system of Flanagan (2006) allows one to write this program without any casts, and uses a theorem prover and a type-directed inference system to add the casts shown above. However, we want to stress the point that a theorem prover, or a fancy inference system, is not essential. The above is not just useful as a core calculus, but is also suitable as a source language.

The type system presented in this paper does not require subtyping or subsumption, unlike similar type systems in the literature (Flanagan 2006; Gronski et al. 2006; Ou et al. 2004). This gives the system the pleasant property of *unicity of type*: every well-typed term has exactly one type. (This constrasts with *principal types*, where every well-typed term has a most general type, of which all its other types are instances.) 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 the above term is not $4 : \texttt{Int}$ but $4_{\textbf{Nat}} : \texttt{Nat}$.

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 on to a subset type reduces to a dynamic test of the condition on the type. Recall that `Nat` denotes $\{x : \texttt{Int} \mid x \geq 0\}$.

[1] This is a revision of a workshop paper (Wadler and Findler 2007). The current version is completely rewritten, has an improved treatment of blame (a rule requiring 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); and a generalized treatment of subset types (previously subset types were limited to base types, whereas now they may be over any type).

Here is a successful test:

$$\begin{array}{l} \quad \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p \; 4 \\ \longrightarrow \\ \quad 4 \geq 0 \;\; \rhd^p \;\; 4_{\textbf{Nat}} \\ \longrightarrow \\ \quad 4_{\textbf{Nat}} \end{array}$$

And here is a failed test:

$$\begin{array}{l} \quad \langle \texttt{Nat} \Leftarrow \texttt{Int} \rangle^p \; -4 \\ \longrightarrow \\ \quad -4 \geq 0 \;\; \rhd^p \;\; -4_{\textbf{Nat}} \\ \longrightarrow \\ \quad \Uparrow p \end{array}$$

The middle line shows a new term form that performs a dynamic test, of the form $s \rhd^p v_{\{x:T|t\}}$. 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.

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

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`.

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

Again, a cast on the function breaks into two casts, each in opposite directions. What is interesting here is the cast on the argument—reduction converts the *static* type `Int` of the argument of f into a *dynamically* enforced cast!

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 type checker guarantees that f returns a value of type `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 two of the examples considered above, with the more precise type on the left, and the less precise type on the right.

$$\begin{array}{l} \texttt{Nat} \to \texttt{Nat} <:_n \texttt{Int} \to \texttt{Int} \\ \texttt{Int} \to \texttt{Int} <:_n \texttt{Dyn} \to \texttt{Dyn} \end{array}$$

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 takes untyped code and gives it a type. Such a cast makes types more precise, and so cannot give rise to negative blame. For instance, the following code fails, blaming p.

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

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, the following code fails, blaming $\bar{p}$.

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

Because the blame is negative, the fault lies with the untyped 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.

The same analysis generalizes to code containing refinement types. For instance, the following code fails, blaming q.

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

Here 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.

We now formalise the above analysis.

3. Types, reduction, subtyping

Findler and Felleisen (2002) includes a system with dependent contracts, and Flanagan (2006) and Ou et al. (2004) similarly use dependent function types. We follow Gronski and Flanagan (2007), in using a simpler system without dependent function types. Refinement types include terms within types and thus constitute a restricted form of dependent type. We omit dependent function types for simplicity. (It is not clear whether it is possible, or desirable, to maintain unicity of types in the presence of dependent function types.)

Flanagan (2006), Ou et al. (2004), and Gronski and Flanagan (2007) restrict subset types to base types. We follow Gronski et al. (2006) in permitting subset types over arbitrary types.

Compile-time type rules of our system are presented in Figure 2, run-time type rules and reduction rules in Figure 3, and rules for subtyping in Figure 4. We discuss each of these in turn in the following three subsections.

3.1 Types and terms

Figure 2 presents the syntax of types and terms and the compile-time type rules. The language is explicitly and statically typed. We discuss how to embed untyped terms in Section 3.4.

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 $\{x : T \mid t\}$. 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 $ty(c)$. 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.

The type system is explained in terms of three judgements, which are presented in Figure 2. We write T *wf* if type T is well formed, 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 $\{x : T \mid t\}$ we have that t has type $\texttt{Bool}$ on the assumption that x has type T (no other free variables may appear in t). In what follows, we assume all types are well-formed. We call T the *domain* of the subset type $\{x : T \mid t\}$.

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 (which we define 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.

Since we do not have subsumption our type system over the source language remains decidable. A pleasant consequence of omitting subsumption is that, as with gradual types, each term has a unique type.

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

An even more pleasant consequence is that our type system for the source language is decidable, unlike that for hybrid types.

Proposition 2. *(Decidability) Given* Γ *and* t*, it is decidable whether there is a* T *such that* $\Gamma \vdash t : T$.

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 3 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 two 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. Flanagan (2006) and Siek and Taha (2006) make 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

$$\texttt{Dyn}_{\texttt{Dyn}\to\texttt{Dyn}}(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int}\rangle^p\ (\lambda x : \texttt{Int}.\ x + 1)).$$

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	$::=$	$\texttt{Dyn} \mid B \mid S \to T \mid \{x : T \mid t\}$
terms	s, t, u	$::=$	$x \mid c \mid \lambda x : S.\ t \mid t\ s \mid \langle T \Leftarrow S \rangle^p\ s$

Well-formed types $\boxed{T\ wf}$

$$\frac{}{\texttt{Dyn}\ wf} \qquad \frac{}{B\ wf} \qquad \frac{S\ wf \qquad T\ wf}{(S \to T)\ wf} \qquad \frac{T\ wf \qquad x : T \vdash t : \texttt{Bool}}{\{x : T \mid t\}\ wf}$$

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

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

Compatibility $\boxed{S \sim T}$

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

Figure 2. Compile-time types

Syntax

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

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

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

Reductions $\boxed{s \longrightarrow r}$

$$\begin{array}{lcll}
c\ v & \longrightarrow & [\![c]\!](v) & \\
(\lambda x : S.\ t)\ v & \longrightarrow & t[x := v] & \\
\langle B \Leftarrow B \rangle^p\ v & \longrightarrow & v & \\
(\langle S' \to T' \Leftarrow S \to T \rangle^p\ v)\ w & \longrightarrow & \langle T' \Leftarrow T \rangle^p\ v\ (\langle S \Leftarrow S' \rangle^{\bar{p}}\ w) & \\
\langle \texttt{Dyn} \Leftarrow \texttt{Dyn} \rangle^p\ v & \longrightarrow & v & \\
\langle \texttt{Dyn} \Leftarrow B \rangle^p\ v & \longrightarrow & \texttt{Dyn}_B(v) & \\
\langle \texttt{Dyn} \Leftarrow S \to T \rangle^p\ v & \longrightarrow & \texttt{Dyn}_{\texttt{Dyn}\to\texttt{Dyn}}(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow S \to T \rangle^p\ v) & \\
\langle T \Leftarrow \texttt{Dyn} \rangle^p\ \texttt{Dyn}_G(v) & \longrightarrow & \langle T \Leftarrow G \rangle^p\ v, & \text{if } G \sim T \\
\langle T \Leftarrow \texttt{Dyn} \rangle^p\ \texttt{Dyn}_G(v) & \longrightarrow & \Uparrow p, & \text{if } G \not\sim T \\
\langle \{x : T \mid t\} \Leftarrow S \rangle^p\ v & \longrightarrow & \texttt{let}\ x = \langle T \Leftarrow S \rangle^p\ v\ \texttt{in}\ t \rhd^p x_{\{x:T\mid t\}} & \\
\texttt{true} \rhd^p v_{\{x:T\mid t\}} & \longrightarrow & v_{\{x:T\mid t\}} & \\
\texttt{false} \rhd^p v_{\{x:T\mid t\}} & \longrightarrow & \Uparrow p & \\
\langle T \Leftarrow \{x : S \mid s\} \rangle^p\ v_{\{x:S\mid s\}} & \longrightarrow & \langle T \Leftarrow S \rangle^p\ v &
\end{array}$$

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \qquad \frac{s \longrightarrow \Uparrow p}{E[s] \longrightarrow \Uparrow p}$$

Figure 3. Run-time types and reduction

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 $v_{\{x:T|t\}}$ where v is a value of type T 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 $s \rhd^p v_{\{x:T|t\}}$, 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 true the term reduces to $v_{\{x:T|t\}}$, and if s reduces to false the term allocates blame to p.[2]

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 2, and fails when these are extended with the run-time type rules of Figure 3. 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. Constants of function type are interpreted by a semantic function consistent with their type: if $ty(c) = S \to T$ and value v has type S, then $[\![c]\!](v)$ is a term of type T.

$$c\, v \quad \longrightarrow \quad [\![c]\!](v)$$

For example, $ty(+) = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$, with $[\![+]\!](3) = +_3$, where $ty(+_3) = \texttt{Int} \to \texttt{Int}$ and $[\![+_3]\!](4) = 7$.

The rule for applying a lambda expression is standard.

$$(\lambda x : S.\ t)\, v \quad \longrightarrow \quad t[x := v]$$

A cast to a base type from itself is the identity.

$$\langle B \Leftarrow B \rangle^p\, v \quad \longrightarrow \quad v$$

A cast to a function type from another function type decomposes into separate casts on the argument and result.

$$(\langle S' \to T' \Leftarrow S \to T \rangle^p\, v)\, w \quad \longrightarrow \quad \langle T' \Leftarrow T \rangle^p\, v\, (\langle S \Leftarrow S' \rangle^{\bar{p}}\, w)$$

Note the reversal in the argument cast, and the corresponding negating of the blame label.

[2] In contrast, Flanagan (2006) has essentially the following rule.

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

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.

The next three rules concern casts to the dynamic type.

$$\begin{array}{lcl} \langle \texttt{Dyn} \Leftarrow \texttt{Dyn} \rangle^p\, v & \longrightarrow & v \\ \langle \texttt{Dyn} \Leftarrow B \rangle^p\, v & \longrightarrow & \texttt{Dyn}_B(v) \\ \langle \texttt{Dyn} \Leftarrow S \to T \rangle^p\, v & \longrightarrow & \\ \multicolumn{3}{r}{\texttt{Dyn}_{\texttt{Dyn}\to\texttt{Dyn}}(\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow S \to T \rangle^p\, v)} \end{array}$$

A cast to Dyn from itself is the identity. A cast to Dyn from a base type is a value. A cast to Dyn from a function type $S \to T$ decomposes into a cast to Dyn from the ground type $\texttt{Dyn} \to \texttt{Dyn}$, and a cast to $\texttt{Dyn} \to \texttt{Dyn}$ from $S \to T$.

The next two rules concern casts from the dynamic type.

$$\begin{array}{lcll} \langle T \Leftarrow \texttt{Dyn} \rangle^p\, \texttt{Dyn}_G(v) & \longrightarrow & \langle T \Leftarrow G \rangle^p\, v, & \text{if } G \sim T \\ \langle T \Leftarrow \texttt{Dyn} \rangle^p\, \texttt{Dyn}_G(v) & \longrightarrow & \Uparrow p, & \text{if } G \not\sim T \end{array}$$

Consider a cast to type T from the dynamic type. Recall that values of the dynamic type have the form $\texttt{Dyn}_G(v)$, where G is a ground type and v has type G. If the types T and G are compatible, the cast collapses to a cast directly to T from G, otherwise the cast fails with blame allocated to the label on the original cast.

The next three rules concern casts to subset type.

$$\begin{array}{lcl} \langle \{x : T \mid t\} \Leftarrow S \rangle^p\, v & \longrightarrow & \\ \multicolumn{3}{r}{\texttt{let}\ x = \langle T \Leftarrow S \rangle^p\, v\ \texttt{in}\ t \rhd^p x_{\{x:T|t\}}} \\ \texttt{true} \rhd^p v_{\{x:T|t\}} & \longrightarrow & v_{\{x:T|t\}} \\ \texttt{false} \rhd^p v_{\{x:T|t\}} & \longrightarrow & \Uparrow p \end{array}$$

A cast to subset type with domain T from type S decomposes into a cast to T from S, followed by a test that the value satisfies the predicate. If the predicate evaluates to true the test reduces to the subset type, otherwise it allocates blame to the label on the test.

The next rule concerns casts from a subset type.

$$\langle T \Leftarrow \{x : S \mid s\} \rangle^p\, v_{\{x:S|s\}} \quad \longrightarrow \quad \langle T \Leftarrow S \rangle^p\, v$$

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

The final two rules give the compatible closure of reduction with regard to evaluation contexts.

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \qquad \frac{s \longrightarrow \Uparrow p}{E[s] \longrightarrow \Uparrow p}$$

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 4 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 result. 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.

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

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

Subtype $\boxed{S <: T}$

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

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

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

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

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

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

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

Figure 4. Subtypes

Entailment, and hence subtyping, are undecidable. This is not a hindrance, since our type system does not depend on subtyping. Defining subtyping in terms of entailment 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 Flanagan (2006), Ou et al. (2004), and Gronski et al. (2006). Our treatment is particularly close to the latter, which include the dynamic type and allows subset types over any domain (the other two restrict subset types to be over base types).

However, Gronski et al. (2006) 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`, as discussed by Findler and Blume (2006).

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.

It is easy to check that all four of the relations $<:$, $<:^+$, $<:^-$, and $<:_n$ are reflexive and transitive. We also have that $S <: T$ implies $S \sim T$, and similarly for the three other relations.

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 function arguments and results. 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 $T <:^- S$.) Hence if S is more precise than T we have $S <:^+ T$, and if S is less precise than T we have $S <:^- T$. This result connects our informal discussion relating precision and blame above to our formal results below.

Here are some examples:

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

The first line shows that ordinary subtyping is contravariant in the domain and covariant in the range, while the fourth line shows that naive subtyping is covariant in both. The first line is equivalent to the second and third, and the fourth line is equivalent to the fifth and sixth.

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). Let M, N range over untyped terms.

$$M, N \quad ::= \quad x \mid k \mid \lambda x.\ N \mid M\ N \mid \lfloor t \rfloor$$

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 `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 Dyn, and if every typed subterm has type Dyn. We write $\Gamma \vdash M\ wf$ to indicate that M is well formed.

$$\frac{(x : \mathtt{Dyn}) \in \Gamma}{\Gamma \vdash x\ wf} \qquad \frac{\Gamma,\ x : \mathtt{Dyn} \vdash N\ wf}{\Gamma \vdash (\lambda x.\ N)\ wf}$$

$$\frac{\Gamma \vdash M\ wf \qquad \Gamma \vdash N\ wf}{\Gamma \vdash (M\ N)\ wf} \qquad \frac{\Gamma \vdash t : \mathtt{Dyn}}{\Gamma \vdash \lfloor t \rfloor\ wf}$$

A simple mapping takes untyped terms into typed terms.

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

An untyped term is well-formed if and only if the corresponding typed term is well-typed with type Dyn.

Lemma 3. *We have $\Gamma \vdash M\ wf$ if and only if $\Gamma \vdash \lceil M \rceil : \mathtt{Dyn}$.*

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

3.5 Type safety

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

Lemma 4. *(Substitution) If $\Gamma \vdash v : S$ and $\Gamma,\ x : S \vdash t : T$, then $\Gamma \vdash t[x := v] : T$.*

Lemma 5. *(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 $ty(c) = S \to T$, or*
  - *$v = \langle S \to T \Leftarrow S' \to T' \rangle^p\ v'$ with $\vdash v' : S' \to T'$.*
- *If $\vdash v : \{x : T \mid t\}$ then $v = v'_{\{x:T|t\}}$ with $\vdash v' : T$ and $t[x := v'] \longrightarrow^* \mathtt{true}$.*
- *If $\vdash v : \mathtt{Dyn}$ then $v = \mathtt{Dyn}_G(v')$ with $\vdash v' : G$.*

Proposition 6. *(Preservation) If $\Gamma \vdash s : T$ and $s \longrightarrow t$ then $\Gamma \vdash t : T$.*

Proof. By induction over type derivations, with one case for each reduction rule. We consider only the unusual cases.

- Consider the reduction
$$\langle \{x : T \mid t\} \Leftarrow S \rangle^p\ v \longrightarrow \mathtt{let}\ x = \langle T \Leftarrow S \rangle^p\ v\ \mathtt{in}\ t \rhd^p x_{\{x:T|t\}}$$
This preserves types because of the run-time typing rule for $s \rhd^p v_{\{x:T|t\}}$ is trivially satisfied when $s = t$ and $v = x$.
- Consider the reduction
$$\mathtt{true} \rhd^p v_{\{x:T|t\}} \longrightarrow v_{\{x:T|t\}}$$
The left-hand side can only be well-typed by the run-time typing rule
$$\frac{\Gamma \vdash s : \mathtt{Bool} \qquad \Gamma \vdash v : T \qquad t[x := v] \longrightarrow^* s}{\Gamma \vdash s \rhd^p v_{\{x:T|t\}} : \{x : T \mid t\}}$$
and we must have $s = \mathtt{true}$, in which case the right-hand side is well typed by the run-time type rule
$$\frac{\Gamma \vdash v : T \qquad t[x := v] \longrightarrow^* \mathtt{true}}{\Gamma \vdash v_{\{x:T|t\}} : \{x : T \mid t\}}.$$
- Consider the reduction
$$\mathtt{false} \rhd^p v_{\{x:T|t\}} \longrightarrow \Uparrow p$$
This does not match the hypothesis, because $\Uparrow p$ is not a term. □

Proposition 7. *(Progress) If $\vdash s : T$ then either*

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

Proof. By induction over type derivations, with one case for each reduction rule. We consider only the unusual cases.

- Consider a derivation ending
$$\frac{\vdash v : T \qquad t[x := v] \longrightarrow^* \mathtt{true}}{\vdash v_{\{x:T|t\}} : \{x : T \mid t\}}.$$
In this case, the typed term is a value.
- Consider a derivation ending
$$\frac{\vdash s : \mathtt{Bool} \qquad \vdash v : T \qquad t[x := v] \longrightarrow^* s}{\vdash s \rhd^p v_{\{x:T|t\}} : \{x : T \mid t\}}$$
Since $\vdash s : \mathtt{Bool}$, by induction there are three possibilities for s.
  - s is a value, in which case s must be true or false, and the term reduces to $v_{\{x:T|t\}}$ or $\Uparrow p$.
  - $s \longrightarrow s'$ for some s', and the term reduces to $s' \rhd^p v_{\{x:T|t\}}$ with
$$\vdash s' \rhd^p v_{\{x:T|t\}} : \{x : T \mid t\}$$
  - $s \longrightarrow \Uparrow q$, for some q, and the term reduces to $\Uparrow q$. □

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.

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

Proof. By induction on the syntactic form of S and T. There are sixteen cases to consider (four syntactic forms for each of S and T), but several of these combine or are trivial. We consider some interesting cases.

- $\mathtt{Dyn} <: \mathtt{Dyn}$
  iff (by definition)
  true
  iff (by definition)
  $\mathtt{Dyn} <:^+ \mathtt{Dyn}$ and $\mathtt{Dyn} <:^- \mathtt{Dyn}$.
- $S \to T <: S' \to T'$
  iff (by definition)
  $S' <: S$ and $T <: T'$
  iff (by induction hypothesis)
  ($S' <:^+ S$ and $S' <:^- S$) and ($T <:^+ T'$ and $T <:^- T'$)
  iff (rearrange)

($S' <:^- S$ and $T <:^+ T'$) and ($S' <:^+ S$ and $T <:^- T'$)
iff (by definition)
$S \to T <:^+ S' \to T'$ and $S \to T <:^- S' \to T'$.

- $S <: \{x : T \mid t\}$
iff (by definition)
$S <: T$ and $x : T \Leftarrow S \models t$
iff (by induction hypothesis)
($S <:^+ T$ and $S <:^- T$) and $x : T \Leftarrow S \models t$
iff (rearrange)
($S <:^+ T$ and $x : T \Leftarrow S \models t$) and $S <:^- T$
iff (by definition)
$S <:^+ \{x : T \mid t\}$ and $S <:^- \{x : T \mid t\}$.

The other cases are similar. □

Naive subtyping also factors into positive and negative subtyping, this time with the direction of negative subtyping reversed. Hence, narrowing implies positive subtyping and widening implies negative subtyping.

Proposition 9. *(Factoring naive subtyping) We have $S <:_n T$ if and only if $S <:^+ T$ and $T <:^- S$.*

Proof. Similar to the previous proof. We consider some interesting cases.

- $S <:_n \texttt{Dyn}$
iff (by definition)
true
iff (by definition)
$S <:^+ \texttt{Dyn}$ and $\texttt{Dyn} <:^- S$.
- $S \to T <:_n S' \to T'$
iff (by definition)
$S <:_n S'$ and $T <:_n T'$
iff (by induction hypothesis)
($S <:^+ S'$ and $S' <:^- S$) and ($T <:^+ T'$ and $T' <:^- T$)
iff (rearrange)
($S' <:^- S$ and $T <:^+ T'$) and ($S <:^+ S'$ and $T' <:^- T$)
iff (by definition)
$S \to T <:^+ S' \to T'$ and $S \to T <:^- S' \to T'$.
- $S <:_n \{x : T \mid t\}$
iff (by definition)
$S <:_n T$ and $x : T \Leftarrow S \models t$
iff (by induction hypothesis)
($S <:^+ T$ and $T <:^- S$) and $x : T \Leftarrow S \models t$
iff (rearrange)
($S <:^+ T$ and $x : T \Leftarrow S \models t$) and $T <:^- S$
iff (by definition)
$S <:^+ \{x : T \mid t\}$ and $\{x : T \mid t\} <:^- S$.

The other cases are similar. □

The following is the central result of this paper. Note that the subterms of a term include any term in a subset type in a cast.

Proposition 10. *(Positive and negative blame) Let t be a well-typed term and p be a blame label, and consider all subterms of t containing p. If*

- *every cast with label p is a positive subtype, $\langle T \Leftarrow S\rangle^p$ s has $S <:^+ T$, and*
- *every cast with label $\bar{p}$ is a negative subtype, $\langle T \Leftarrow S\rangle^{\bar{p}}$ s has $S <:^- T$, and*
- *every predicate test with label p will succeed, $s \rhd^p v_{\{x:T|t\}}$ has $s \longrightarrow^* \texttt{true}$*

then $t \not\longrightarrow^* \Uparrow p$.

Proof. By induction over the length of the reduction sequence. For each reduction, we show that if every subterm with a blame label on the left-hand side satisfies the hypothesis, so does every subterm with a blame label on the right-hand side. A trivial note: the hypothesis puts no constraint on predicate tests with label $\bar{p}$, and the proposition does not rule out reduction to $\Uparrow \bar{p}$.

- Consider the reduction
$$(\langle S' \to T' \Leftarrow S \to T\rangle^p\ v)\ w \quad \longrightarrow \quad \langle T' \Leftarrow T\rangle^p\ v\ (\langle S \Leftarrow S'\rangle^{\bar{p}}\ w)$$
If $S \to T <:^+ S' \to T'$ for the cast on the left, then $S' <:^- S$ and $T <:^+ T'$ for the casts on the right, by definition of $<:^+$. Similarly, if $S \to T <:^- S' \to T'$ for the cast on the left, then $S' <:^+ S$ and $T <:^- T'$ for the casts on the right, by definition of $<:^-$.
- Consider the reduction
$$\langle \{x : T \mid t\} \Leftarrow S\rangle^p\ v \quad \longrightarrow \quad \texttt{let}\ x = \langle T \Leftarrow S\rangle^p\ v\ \texttt{in}\ t \rhd^p x_{\{x:T|t\}}$$
If $S <:^+ \{x : T \mid t\}$ for the cast on the left, then $S <:^+ T$ for the cast on the right. Further, $x : T \Leftarrow S \models t$ implies $t \longrightarrow^* \texttt{true}$, so the hypothesis also holds for the test on the right.
Conversely, if $S <:^- \{x : T \mid t\}$ for the cast on the left, then $S <:^- T$ for the cast on the right, and the predicate test satisfies the hypothesis trivially (see 'a trivial note' above).
- Consider the reduction
$$\langle T \Leftarrow \texttt{Dyn}\rangle^p\ \texttt{Dyn}_G(v) \quad \longrightarrow \quad \langle T \Leftarrow G\rangle^p\ v, \quad \text{if } G \sim T$$
If $\texttt{Dyn} <:^+ T$ holds for the cast on the left, then $G <:^+ T$ holds for the cast on the right, since $G <:^+ \texttt{Dyn}$ and $<:^+$ is transitive. Similarly, if $\texttt{Dyn} <:^- T$ holds for the cast on the left, then $G <:^- T$ holds for the cast on the right, since $G <:^- \texttt{Dyn}$ and $<:^-$ is transitive.
- Consider the reduction
$$\langle T \Leftarrow \texttt{Dyn}\rangle^p\ \texttt{Dyn}_G(v) \quad \longrightarrow \quad \Uparrow p, \quad \text{if } G \not\sim T$$
If $\texttt{Dyn} <:^+ T$ then either $T = \texttt{Dyn}$ or $T = \{x : T' | t\}$ with $\texttt{Dyn} <: T'$. It follows immediately that $G \sim T$ must hold. So the reduction can never apply when $\texttt{Dyn} <:^+ T$.
Conversely, $\texttt{Dyn} <:^- T$ always holds, but the hypothesis is trivially satisfied (see 'a trivial note' above.) □

We have an immediate corollary.

Corollary 11. *(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. Applications

5.1 Siek and Taha

Siek and Taha (2006) describe an intermediate language similar to the one described here: it is decidable, has compatibility but no subtyping, and possesses unicity of type. The type we write as `Dyn` they write as '`?`', and the cast we write as $\langle T \Leftarrow S \rangle^{pn}\ s$ they write as $\langle T \rangle\ s$. (Given unicity of type, the type S of term s in the cast is redundant.)

Siek and Taha present two languages, a source language and an intermediate language, and a compilation algorithm that takes the first into the second, inserting casts to convert to and from the dynamic type.

They show that if the original program is well-typed in simply-typed lambda calculus then it is well-typed in their system, and they show that the only way an evaluation can go wrong is if some cast fails. It follows that any intermediate program derived from a well-typed source program with no dynamic types cannot get stuck.

But theirs is an all-or-nothing result. Our results show that if a cast fails in a gradually typed program, then the blame must lie with a fragment of the program that contains a dynamic type. Since the purpose of gradual typing is to permit dynamic types in programs, our result is a useful supplement to theirs.

5.2 Flanagan

Flanagan (2006) describes an intermediate language similar to the language described here, except that it includes subsumption; hence the type system is undecidable and does not have unicity of type. The cast we write as $\langle T \Leftarrow S \rangle^p\ s$ he writes as $\langle S \lhd T \rangle\ s$.

Flanagan presents two languages, a source language and an intermediate language, and a compilation algorithm that takes the first into the second, inserting casts to check inclusion between types when a theorem prover fails to show that one is a subtype of the other.

He shows that the compilation algorithm inserts only upcasts when the original program is well-typed, and that in this case the compiled program yields the same result as the original program. It follows that an intermediate program that is derived from a well-typed source program does not get stuck.

But his is an all-or-nothing result. Our results show that any upcast in the intermediate language (that is, a cast from S to T where $S <: T$) does not get stuck, even if the intermediate program contains other casts that are not upcasts. Since the purpose of hybrid types is to insert dynamic checks when the theorem prover fails to prove a subtyping relation (or to find a counterexample), our result is a useful supplement to his.

5.3 Matthews and Findler

Matthews and Findler (2007) set out to solve a different problem than the one we tackle here, but the technical details turn out to be surprisingly similar.

Matthews and Findler define cross-language casts between Scheme and ML. If `e` is a term in Scheme, they write

$$^{\tau}MSG\ \texttt{e}$$

for the corresponding term of type τ in ML. Further, they prove that this conversion need only check for positive blame. The corresponding conversion in our calculus is

$$\langle T \Leftarrow \texttt{Dyn} \rangle^p\ t$$

where T corresponds to τ and t corresponds to `e`. Since $\texttt{Dyn} <:^- T$ for any type T, we know that negative blame cannot arise, which explains why Matthews and Findler only check for positive blame.

Similarly, if **e** is a term of type τ in ML, they write

$$GSM^{\tau}\ \mathbf{e}$$

for the corresponding term in Scheme. Further, they prove that this conversion need only check for negative blame. The corresponding conversion in our calculus is

$$\langle \texttt{Dyn} \Leftarrow T \rangle^p\ t$$

where T corresponds to τ and t corresponds to **e**. Since $T <:^+ \texttt{Dyn}$ for any type T, we know that positive blame cannot arise, which explains why Matthews and Findler only check for negative blame.

Matthews and Findler were concerned with integrating two independently specified programming languages, whereas we are concerned with adding casts to a single language. But it turns out their results are easily handled in our framework. Instead of treating idealized Scheme and ML as separate languages, we have a single language into which both are trivially embedded.

Our approach simplifies definitions and proofs, since we need consider roughly half as many cases (only one form each of variables, constants, abstraction, and application, rather than separate typed and untyped versions). Further, Matthews and Findler in effect only support casts where either the source or target is `Dyn`, while we support casts between any two types.

However, without the Matthews and Findler work it would not be so clear that our work does indeed provide a good model of integrating two languages. Further, their approach may offer a better basis than ours for extension to more complex languages, in cases where the embedding into a single language is less clear.

5.4 Gronski and Flanagan

Gronski and Flanagan (2007) relate the contracts of Findler and Felleisen (2002), modeled as a calculus λ^C, to the hybrid types of Flanagan (2006), modeled as a calculus λ^H, by giving a translation of λ^C into λ^H that preserves types and reductions.

Their calculus λ^C is a simply typed lambda calculus, with function types and base types only, plus an operation that associates a contract and blame labels to a term. They write this $t^{c,l,l'}$, where c is a contract and l and l' are blame labels. We'll write it as $t^{T,p,\bar{p}}$, since our types (including subset types) correspond essentially to their contracts, and we use a single label (which can be negated) where they use pairs. In the phrase $t^{T,p,\bar{p}}$, the type of t must be $\mathit{base}(T)$, which replaces each subset type by its domain.

$$\begin{array}{lcl} \mathit{base}(B) & = & B \\ \mathit{base}(S \to T) & = & \mathit{base}(S) \to \mathit{base}(T) \\ \mathit{base}(\{x : T \mid t\}) & = & \mathit{base}(T) \end{array}$$

Just as our calculus negates the blame label in contravariant position when applying a function cast, their λ^C swaps the blame labels in contravariant position when applying a function contract.

Their calculus λ^H is essentially like the one given here, with the crucial difference that they have a single blame label. They perform no operation analgous to negating labels as in our calculus, or swapping labels as in λ^C. (They also don't have the type dynamic, and restrict subset types to be over base type.)

The essence of their translation is to take the λ^C term

$$t^{T,p,\bar{p}}$$

into the λ^H term

$$\langle \mathit{base}(T) \Leftarrow T \rangle^{\bar{p}}\ \langle T \Leftarrow \mathit{base}(T) \rangle^p\ t.$$

This is easily understood in terms of our results. Clearly, $T <:_n \mathit{base}(T)$. Hence, $T <:^+ \mathit{base}(T)$, so the leftmost cast can only

allocate negative blame, and $\mathit{base}(T) <:^- T$, so the rightmost cast can only allocate positive blame. Thus their translation manages to assign blame appropriately even though they fail to negate or swap blame labels.

However, in general λ^H, like our language, contains casts of the form $\langle T \Leftarrow S \rangle^p\ s$ where neither $S <:^+ T$ nor $S <:^- T$ holds, so we would argue that negating (or swapping) blame labels is important. If blame is to be allocated, it should be divided into positive blame and negative blame. One doesn't want to know merely which cast has failed, but also whether it is the contained term or the containing context which is to blame for that failure.

6. Related work

6.1 Contracts

The notion of dynamic testing of specifications goes back at least to Parnas (1972). A software engineering strategy based on such checking, as well as the term *contract*, was popularised by the language Eiffel (Meyer 1988).

Findler and Felleisen (2002) introduced the use of higher-order contracts with blame in functional programming.

Blume and McAllester (2006) describe some counterintuitive properties of contracts. Findler and Blume (2006) uses projections to model contracts, and suggests that the counterintuitive aspects of contracts may not be so counterintuitive after all. The issues involved are similar to our discussion of the type Dynamic, and our work echos the (perhaps counterintuitive) observation that one should *not* regard all types as subtypes of Dynamic.

Meunier et al. (2006) is concerned with integrating static and dynamic checking of contracts across modules, where the static checking is implemented as a set-based constraint analysis. Tobin-Hochstadt and Felleisen (2006) is also concerned with integrating static and dynamic checking of contracts across modules, this time using a more traditional type inference algorithm augmented to insert contracts where appropriate. We believe the system presented here provides roughly the same power as these other systems, but in a simpler way.

Gray et al. (2005) discuss the practice of using contracts to interface Java to Scheme, and Matthews and Findler (2007) discuss the theory of using contracts to interface ML to Scheme. The relation of the latter to our work is discussed in Section 5.3.

6.2 Gradual types

Integrating static and dynamic typing is not a new idea, and previous work includes the type Dynamic of Abadi et al. (1991), the soft types of Wright and Cartwright (1997), the partial types of Thatte (1988), and the Scheme-to-ML translation of Henglein and Rehof (1995).

Siek and Taha (2006) introduced gradual types; its relation to our work is described in Section 5.1. Siek and Taha (2007) extends gradual typing to an object-oriented language.

Findler and Felleisen (2002) observed that adding contracts to a program can lose the benefits of tail-recursion, and the same observation applies to gradual types and hybrid types, which both apply a form of contracts. Herman et al. (2007) observes how to restore a bounded-space implementation of tail recursion for gradual types. That work exhibits a further connection between gradual types and hybrid types, since it was performed by the team working on hybrid types. Unfortunately, the techniques in that paper apply only to gradual types, and it is not yet clear how to extend them to hybrid types.

6.3 Hybrid types

Subset types were first introduced in type theory by Nordström and Petersson (1983) and Smith and Salvesen (1988). The form of subset types used in hybrid types was influenced by the refinement types of Freeman and Pfenning (1991) and the Dependent ML of Xi and Pfenning (1999). An embedding of particular subset types (non-empty lists, index ranges) into Haskell or O'Caml is described by Kiselyov and chieh Shan (2006).

Flanagan (2006) introduced hybrid types; its relation to our work is discussed in Section 5.2.

Gronski et al. (2006) describe Sage, a practical language based on hybrid types. Both the theory of Sage and practical experience with it is described. Sage extends hybrid types to subsume gradual types by adding a Dynamic type, written $*$. Further, Sage extends to higher-order types, where $*$ is also the type of all types.

Knowles and Flanagan (2007) present a type reconstruction algorithm for hybrid types that finds principle typings, analogous to Hindley-Milner type reconstruction.

Gronski and Flanagan (2007) investigate the relationship between hybrid types and contracts; its relation to our work is discussed in Section 5.4.

Ou et al. (2004) present a language with dynamically-checked dependent types, which is closely related to the work on hybrid types. There is a compilation from a source language into an intermediate language, very similar to that for hybrid types. The system explicitly labels which portions of the code are to be dynamically checked and which are to be statically checked, similar to our use of the notation $\lceil M \rceil$ to embed untyped lambda calculus.

Both Flanagan (2006) and Ou et al. (2004) support dependent function types, while our work here is restricted to ordinary function types. Also, they both restrict subset types to be over base types, while we follow Gronski et al. (2006) in permitting subset types over any type.

7. Conclusion

For the future, we plan to investigate how to extend our work to support polymorphic types and dependent function types.

Contracts for polymorphic types are considered by Guha et al. (2007) and Matthews and Ahmed (2008). In these works, a cast from a function to a polymorphic type introduces a seal that wraps arguments and unwraps results of the function. The seal prevents the function from examining values corresponding to the quantified type variable, thus guaranteeing semantic parametricity, as proved in the second paper using a step-indexed logical relation. Matthews and Ahmed use the two-language framework of Matthews and Findler (2007), and we are collaborating with them to adapt their results to the single-language framework we use here, yielding the advantages discussed in Section 5.3 (in particular, we expect the new proofs to require roughly half as many cases).

Dependent function types are considered by Flanagan (2006) and Ou et al. (2004). In these works, function types are written $x : S \to T$ (or, equivalently, $\Pi x : S.\ T$) where type T may contain individual variable x. The relevant type rules are as follows.

$$\frac{\Gamma,\ x : S \vdash t : T}{\Gamma \vdash \lambda x : S.\ t : (x : S \to T)}$$

$$\frac{\Gamma \vdash s : (x : S \to T) \qquad \Gamma \vdash t : S}{\Gamma \vdash s\ t : T[x := t]}$$

While extension to dependent types should be possible, there are a few technical challenges. The application rule substitutes a term, not a value, for a variable, and so some care may be required with respect to effects, including raising blame. Also, dependent function types are usually accompanied by a liberal notion of type equivalence, and so it may be desirable to abandon unicity of type.

A natural next step is to combine both extensions, giving a system with universal quantification over both types and individuals

(yielding polymorphic types and dependent function types, respectively), similar in power to type theories such as Coq, LF, or the lambda cube (Barendregt 1992). The presence of casts means that not every term can be guaranteed to normalize, which is crucial if typed terms are to represent proofs. One solution may be to add an effect system: if casts and fixpoints are regarded as impure, then pure terms can be guaranteed to have no side effects and to terminate. Extended in this way, the blame calculus would support a wide spectrum of expressiveness, ranging from Scheme to ML to Coq.

Acknowledgements

This paper benefited enormously from conversations with John Hughes. We also thank Samuel Bronson, Matthias Felleisen, Cormac Flanagan, Oleg Kiselyov, and six anonymous referees.

References

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

Henk Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon, 1992.

Matthias Blume and David McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4&5):375–414, 2006.

Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

Robby Findler and Matthias Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, April 2006.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming (ICFP)*, October 2002.

Cormac Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1991.

Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine grained interoperability through mirrors and contracts. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2005.

Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, April 2007.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Workshop on Scheme and Functional Programming*, September 2006.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*, October 2007.

Robert Harper. *Practical Foundations for Programming Languages*. 2007. Working Draft.

Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.

Dave Herman and Cormac Flanagan. Status report: Specifying JavaScript with ML. In *ACM SIGPLAN Workshop on ML*, October 2007.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, April 2007.

Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. In *Programming Languages meets Program Verification (PLPV)*, August 2006.

Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *European Symposium on Programming (ESOP)*, March 2007.

Jacob Matthews and Amal Ahmed. Parametric polymorphism through runtime sealing. In *European Symposium on Programming (ESOP)*, March 2008.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2007.

Erik Meijer. Static typing where possible, dynamic typing where needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

Bengt Nordström and Kent Petersson. Types and specifications. In *International Federation for Information Processing World Computer Congress (IFIP)*, 1983.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, August 2004.

David L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Workshop on Scheme and Functional Programming*, September 2006.

Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007.

Jan Smith and Anne Salvesen. The strength of the subset type in Martin-Löf's set theory. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1988.

Satish Thatte. Type inference with partial types. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, October 2006.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, September 2007.

Andrew K. Wright and Robert Cartwright. A practical soft typing system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1999.