Well-typed programs can't be blamed

Philip Wadler
University of Edinburgh

Robert Bruce Findler
University of Chicago

Abstract

We show how *contracts* with blame fit naturally with recent work on *hybrid types* and *gradual types*. Unlike hybrid types or gradual types, we require casts in the source code, in order to indicate where type errors may occur. Two (perhaps surprising) aspects of our approach are that refined types can provide useful static guarantees even in the absence of a theorem prover, and that type *dynamic* should not be regarded as a supertype of all other types. We factor the well-known notion of subtyping into new notions of positive and negative subtyping, and use these to characterise where positive and negative blame may arise. Our approach sharpens and clarifies some recent results in the literature.

1. Introduction

Recently, a number of researchers have suggested ways to integrate static and dynamic typing into a single framework. These include the *contracts* of Findler and Felleisen (2002) and others, the *gradual types* of Siek and Taha (2006), and the *hybrid types* of Flanagan (2006) and others. Interfaces between Scheme and statically typed languages have been explored by Gray et al. (2005), Tobin-Hochstadt and Felleisen (2006), and Matthews and Findler (2007). Static and dynamic typing are both supported in Visual Basic (Meijer 2004), with similar integration planned for Perl 6 and Javascript; and one of the designers of Java has argued that static types should be optional (Bracha 2004).

We provide a uniform view of recent work on *contracts*, *gradual types*, and *hybrid types* by introducing a notion of blame (from contracts) to a type system with casts (similar to intermediate languages used for gradual and hybrid types), yielding a system that we call *evolutionary types*. Programmers using this type system may add contracts to evolve dynamically typed programs into statically typed programs (as with gradual types) or to evolve statically typed programs into programs with refinement types (as with hybrid types).

We suggest that what has been used as an intermediate type system for gradual and hybrid types is itself useful as a source language—this has the advantage that it is obvious reading the source language where static guarantees hold and where dynamic checks are enforced. We also suggest, in contrast to 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! Finally, we suggest that one should *not* regard every type as a subtype of the dynamic type.

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. We show how our theorem sharpens the published results for gradual and hybrid types, and clarifies other recent results.

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.

We make the following contributions:

- We introduce our language, showing that a language with explicit casts and no theorem prover (and a little syntactic sugar) 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. 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 Gronski and Flanagan (2007) and Matthews and Findler (2007) (Section 5).

Section 6 describes related work, and Section 7 concludes.

2. Evolutionary Programming

2.1 From Untyped to Typed

Consider the following program written without types.

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

Workshop on Scheme and Functional Programming 30 September 2007, Freiburg, Germany

By default, our programming language is typed, so we escape to an untyped language by surrounding the code with ceiling brackets $\lceil \cdot \rceil$. 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 $\lceil 4 \rceil : \textbf{Dyn}$.

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 in which x and h and the application of h to f are written in a typed language, but the body of f is written untyped and cast to a suitable type. Of course, this isn't very helpful for such a short piece of code, but it should be clear how this would work in a larger system.

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

Here, $\lceil \lambda y.\, y + 1 \rceil$ has type **Dyn** (the type of untyped code), and the cast converts it to type **Int** $\to$ **Int**. The above term evaluates to $4 : \textbf{Int}$.

In general, a cast from source type S to target type T is written

$$\langle T \Leftarrow S \rangle^{pn}\ s,$$

where subterm s has type S and the whole term has type T. The two labels on a cast, p and n, are used for allocating *positive blame* and *negative blame* respectively. Blame labels are simply identifiers, without further structure. Positive blame is allocated if the term contained in the cast fails to satisfy the contract implied by the cast, while negative blame is allocated if 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. Even writing the target can be cumbersome. Both the gradual type system of Siek and Taha (2006) and the hybrid type system of Flanagan (2006) include source languages where most or all casts are omitted, but inferred by a type-directed translation. Our notation is inspired by that of Flanagan (2006), and identical to that of 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 refinement type is $\{x : \textbf{Int} \mid x \geq 0\}$, 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. As the name implies, refinement types are types, so **Int**, **Nat**, **Int** $\to$ **Int** and **Nat** $\to$ **Nat** are all examples of types.

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.

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

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 in the above. However, we want to stress the point that a theorem prover, or even a fancy inference system, is not essential.

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 *principle 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 label constants with their type. Thus the value of the above term is not $4 : \textbf{Int}$ but $4_{\textbf{Nat}} : \textbf{Nat}$. Subscripted constants are used only to explain how evaluation works; in the source program, the user always creates values of refinement types by casts that dynamically check the predicate of that type. For instance, above we wrote $\langle \textbf{Nat} \Leftarrow \textbf{Int} \rangle^{pn}\ 2$ which evaluates to $2_{\textbf{Nat}} : \textbf{Nat}$.

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 a refinement type reduces to a dynamic test of the condition on the type.

$$\begin{array}{l}\quad \langle \textbf{Nat} \Leftarrow \textbf{Int} \rangle^{pn}\ (-2)\\ \longrightarrow\\ \quad \textbf{if } -2 \geq 0 \textbf{ then } -2_{\textbf{Nat}} \textbf{ else blame } p\\ \longrightarrow\\ \quad \textbf{blame } p\end{array}$$

(The middle term in this reduction may not appear to be well-typed, but the type system has a special rule for just this purpose, to assure that reductions preserve types.)

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.

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

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 labels on the reversed cast have been swapped from pn to np. The blame labels are swapped on the argument cast because if that cast fails it is the fault of the context, which supplies the argument to the function; swapping moves the negative label to the positive position, so it will take the blame if something goes wrong. Conversely, the blame label is not swapped 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**.

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

Again, a cast on the function breaks into two casts, each in opposite directions. What is interesting here is that 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

$$(\langle \textbf{Nat} \to \textbf{Nat} \Leftarrow \textbf{Int} \to \textbf{Int} \rangle^{pn}\ f)\ x$$

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 $\lambda y : \textbf{Int}.\ y - 2$ and x is 1.

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

$$(\langle \textbf{Dyn} \to \textbf{Dyn} \Leftarrow \textbf{Int} \to \textbf{Int} \rangle^{pn}\ f)\ x$$

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 $\lambda y : \textbf{Int}.\ y + 1$ and x is $\lceil \textbf{true} \rceil$.

One contribution of this paper is that we will characterise those situations in which we can ensure that negative or positive blame cannot arise. Roughly speaking, if a cast is making a type more precise it cannot give rise to negative blame, while if it is making a type less precise it cannot give rise to positive blame.

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 the cast with the label p.

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

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 n.

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

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 faults. The point is that 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 p'.

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

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

We now begin the formal development of our work.

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. Extending to dependent function types should be straightforward, but we leave this for future work.

We also follow Flanagan (2006) and Gronski and Flanagan (2007) in restricting subset types to base types and treating base types as a special case of subset types, which is technically simpler. Gronski et al. (2006) permits subsets over arbitary types.

Compile-time type rules of our system are presented in Figure 1, reduction rules in Figure 2, additional run-time type rules in Figure 3, and rules for subtyping in Figure 4. We discuss each of these in turn in the following four subsections.

3.1 Types and terms

Figure 1 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 3.6.

We let S, T range over types, and s, t range over terms. A type is either a function type $S \to T$; a subset type $\{x : B \mid t\}$, where B is a base type, and t is a term of type **Bool** with a free variable x of type B; or the dynamic type **Dyn**. In types we may write B as an abbreviation for $\{x : B \mid \textbf{true}\}$.

A term is either a variable x; a constant c; a conditional expression **if** s **then** t **else** u; a lambda expression $\lambda x : S.\ t$; an application $s\ t$; or a cast expression $\langle T \Leftarrow S \rangle^{pn}\ s$.

The type system is explained in terms of three related judgements, which are presented in Figure 1. We write $\Gamma \vdash t : T$ if term t has type T in environment Γ, we write $\Gamma \vdash T$ **wf** if type T is well formed in environment Γ, and we write Γ **wf** if environment Γ is well formed. It is easy to check that $\Gamma \vdash t : T$ implies $\Gamma \vdash T$ **wf**, and $\Gamma \vdash T$ **wf** implies Γ **wf**.

We assume a denumerable set of constants. Every constant c is assigned a unique type $\mathrm{ty}(c)$, which must be either a base type B or a function type $S \to T$. We assume **Bool** is a base type with **true** and **false** as constants of type **Bool**; and that **Int** is a base type with 0, 1, and so on, as constants of type **Int**, and $+$ and $-$ as constants

Syntax

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

Type rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \textbf{ wf} \qquad (x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \textbf{ wf} \qquad T = \mathrm{ty}(c)}{\Gamma \vdash c : T} \qquad \frac{\Gamma \vdash s : \textbf{Bool} \qquad \Gamma \vdash t : T \qquad \Gamma \vdash u : T}{\Gamma \vdash (\textbf{if } s \textbf{ then } t \textbf{ else } u) : T}$$

$$\frac{\Gamma \vdash S \textbf{ wf} \qquad \Gamma \vdash T \textbf{ wf} \qquad \Gamma,\, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S.\, t) : (S \to T)} \qquad \frac{\Gamma \vdash t : (S \to T) \qquad \Gamma \vdash s : S}{\Gamma \vdash (t\ s) : T} \qquad \frac{\Gamma \vdash s : S \qquad \Gamma \vdash T \textbf{ wf} \qquad S \sim T}{\Gamma \vdash (\langle T \Leftarrow S \rangle^{pn}\ s) : T}$$

Well-formed types $\boxed{\Gamma \vdash T \textbf{ wf}}$

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

Well-formed context $\boxed{\Gamma \textbf{ wf}}$

$$\frac{}{\emptyset \textbf{ wf}} \qquad \frac{\Gamma \textbf{ wf} \qquad \Gamma \vdash T \textbf{ wf}}{\Gamma,\, x : T \textbf{ wf}}$$

Compatibility $\boxed{S \sim T}$

$$\frac{S \sim S' \qquad T \sim T'}{(S \to T) \sim (S' \to T')} \qquad \frac{}{\{x : B \mid s\} \sim \{x : B \mid t\}} \qquad \frac{}{S \sim \textbf{Dyn}} \qquad \frac{}{\textbf{Dyn} \sim T}$$

Figure 1. Type system

Syntax

terms	s, t, u	$::=$	$\cdots \mid c_T \mid$ **if** s **then** c_T **else blame** p
values	v, w	$::=$	$c \mid c_T \mid \lambda x : S.\, t \mid \langle S' \to T' \Leftarrow S \to T \rangle^{pn}\ v \mid \langle$**Dyn** $\Leftarrow S \rangle^{pn}\ v$
results	r	$::=$	$t \mid$ **blame** p
evaluation context	E	$::=$	$[\,] \mid$ **if** E **then** t **else** $u \mid$ **if** E **then** c_T **else blame** $p \mid E\ t \mid v\ E \mid \langle T \Leftarrow S \rangle^{pn}\ E$

Reductions $\boxed{s \longrightarrow r}$

$c\ v$	$\longrightarrow$	$[\![c]\!](v)$	
$(\lambda x.\, t)\ v$	$\longrightarrow$	$t[x := v]$	
if true then t **else** u	$\longrightarrow$	t	
if false then t **else** u	$\longrightarrow$	u	
$(\langle S' \to T' \Leftarrow S \to T \rangle^{pn}\ v)\ w$	$\longrightarrow$	$\langle T' \Leftarrow T \rangle^{pn}\ (v\ (\langle S \Leftarrow S' \rangle^{np}\ w))$	
$\langle T \Leftarrow S \rangle^{pn}\ c_S$	$\longrightarrow$	**if** $t[x := c_B]$ **then** c_T **else blame** p,	if $S = \{x : B \mid s\}, T = \{x : B \mid t\}$
$\langle T \Leftarrow$ **Dyn** $\rangle^{p'n'}\ (\langle$**Dyn** $\Leftarrow S \rangle^{pn}\ v)$	$\longrightarrow$	$\langle T \Leftarrow S \rangle^{p'n}\ v$,	if $S \sim T$
$\langle T \Leftarrow$ **Dyn** $\rangle^{p'n'}\ (\langle$**Dyn** $\Leftarrow S \rangle^{pn}\ v)$	$\longrightarrow$	**blame** p',	if $S \not\sim T$
if true then c_T **else blame** p	$\longrightarrow$	c_T	
if false then c_T **else blame** p	$\longrightarrow$	**blame** p	
$E[s]$	$\longrightarrow$	$E[t]$	if $s \longrightarrow t$
$E[s]$	$\longrightarrow$	**blame** p	if $s \longrightarrow$ **blame** p

Figure 2. Reduction

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

$$\frac{T = \{x : B \mid t\} \qquad \mathrm{ty}(c) = B \qquad \Gamma \vdash T \textbf{ wf} \qquad \Gamma \models \textbf{true} \Rightarrow t[x := c]}{\Gamma \vdash c_T : T} \qquad \frac{\Gamma \vdash s : \textbf{Bool} \qquad T = \{x : B \mid t\} \qquad \mathrm{ty}(c) = B \qquad \Gamma \vdash T \textbf{ wf} \qquad \Gamma \models s \Rightarrow t[x := c]}{\Gamma \vdash (\textbf{if } s \textbf{ then } c_T \textbf{ else blame } p) : T}$$

Implication $\boxed{\Gamma \models s \Rightarrow t}$

$$\frac{\Gamma \vdash s : \textbf{Bool} \qquad \Gamma \vdash t : \textbf{Bool} \qquad \text{for all } \sigma \text{ such that } \Gamma \models \sigma, \text{ if } \sigma(s) \longrightarrow^* \textbf{true} \text{ then } \sigma(t) \longrightarrow^* \textbf{true}}{\Gamma \models s \Rightarrow t}$$

Consistent substitution $\boxed{\Gamma \models \sigma}$

$$\frac{}{\emptyset \models \emptyset} \qquad \frac{\Gamma \models \sigma \qquad \Gamma \vdash v : T}{\Gamma,\, x : T \models (\sigma,\, x := v)}$$

Figure 3. Run-time type rules

Subtype $\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \textbf{ wf}}{\Gamma \vdash \textbf{Dyn} <: \textbf{Dyn}} \qquad \frac{\Gamma \vdash S' <: S \qquad \Gamma \vdash T <: T'}{\Gamma \vdash (S \to T) <: (S' \to T')} \qquad \frac{\Gamma,\, x : B \models s \Rightarrow t}{\Gamma \vdash \{x : B \mid s\} <: \{x : B \mid t\}}$$

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

$$\frac{\Gamma \vdash S \textbf{ wf}}{\Gamma \vdash S <:^+ \textbf{Dyn}} \qquad \frac{\Gamma \vdash S' <:^- S \qquad \Gamma \vdash T <:^+ T'}{\Gamma \vdash (S \to T) <:^+ (S' \to T')} \qquad \frac{\Gamma,\, x : B \models s \Rightarrow t}{\Gamma \vdash \{x : B \mid s\} <:^+ \{x : B \mid t\}}$$

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

$$\frac{\Gamma \vdash T \textbf{ wf}}{\Gamma \vdash \textbf{Dyn} <:^- T} \qquad \frac{\Gamma \vdash S' <:^+ S \qquad \Gamma \vdash T <:^- T'}{\Gamma \vdash (S \to T) <:^- (S' \to T')} \qquad \frac{\Gamma \vdash \{x : B \mid s\} \textbf{ wf} \qquad \Gamma \vdash \{x : B \mid t\} \textbf{ wf}}{\Gamma \vdash \{x : B \mid s\} <:^- \{x : B \mid t\}}$$

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

$$\frac{\Gamma \vdash S \textbf{ wf}}{\Gamma \vdash S <:_n \textbf{Dyn}} \qquad \frac{\Gamma \vdash S <:_n S' \qquad \Gamma \vdash T <:_n T'}{\Gamma \vdash (S \to T) <:_n (S' \to T')} \qquad \frac{\Gamma,\, x : B \models s \Rightarrow t}{\Gamma \vdash \{x : B \mid s\} <:_n \{x : B \mid t\}}$$

Figure 4. Subtypes and Implication

of type **Int** $\to$ **Int** $\to$ **Int**, and possibly other constants. We assume that the type assigned to each constant is well formed in the empty context.

Conditional expressions and lambda abstraction are as normal. So is application, save the additional constraint $\Gamma \vdash T$ **wf** is required to ensure that x does not appear free in T. The cast rule is straightforward: it says that if term s has type S and T is a well-formed type compatible with S (where compatibility is defined below), then $\langle T \Leftarrow S \rangle^{pn}\ s$ has type T.

We write $S \sim T$ for the *compatibility* relation, which holds if it may be sensible to cast S to T. Two function types are compatible if their domains and ranges are compatible, two subset types are compatible if they have the same base type, and **Dyn** is compatible with every type.

Compatibility is reflexive and symmetric but not transitive. For example, $S \sim$ **Dyn** and **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 subset type, or if S and T are subset types over different base types.

Our cast rule is inspired by the similar rules found for gradual types and hybrid types. Gradual types introduce compatibility and the idea that all types are compatible with the dynamic type, 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 $t[x := c] \longrightarrow^*$ **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 undecidable run-time rules to check the types of these terms. We explain the details of how this works below.

3.2 Reductions

Figure 2 defines values and evaluation contexts, and presents the rules for reduction.

We extend the syntax of terms with two new forms. If c is a constant with $\mathrm{ty}(c) = B$, and T is a subset type $\{x : B \mid t\}$, and we are in a context where $t[x \leftarrow c] \longrightarrow^* \textbf{true}$, then we write c_T to stand for a constant of type T. Subscripting a constant with its type is necessary to ensure that each term has a unique type in the presence of subset types. We write c as an abbreviation for c_B, where B is in turn an abbreviation for $\{x : B \mid \textbf{true}\}$. We also add the term form $\textbf{if}\, s\, \textbf{then}\, c_T\, \textbf{else blame}\, p$. Note that $\textbf{if}\, s\, \textbf{then}\, t\, \textbf{else}\, u$ and $\textbf{if}\, s\, \textbf{then}\, c_T\, \textbf{else blame}\, p$ are distinct terms; they cannot be confused becase **blame** p is not a legal term.

We let v, w range over values. A value is either a constant (subscripted with its type), a lambda expression, a cast from function type to function type, or a cast to a dynamic type.

A value of function type is either a lambda expression $\lambda x : S.\, t$ or a constant of function type c, or a cast applied to a function type, $\langle S' \to T' \Leftarrow S \to T \rangle^{pn}\, v$, where v has type $S \to T$.

A value of subset type $T = \{x : B \mid t\}$ is a constant of the form c_T, where $t[x := c] \longrightarrow^* \textbf{true}$.

A value of dynamic type **Dyn** is a cast of the form $\langle \textbf{Dyn} \Leftarrow S \rangle^{pn}\, v$, where v has type S.

We let E range over evaluation contexts, which are standard. The cast operation is strict, and must reduce the term being cast to a value before the cast can be performed.

We write $s \longrightarrow t$ to indicate that a single reduction step takes term s to term t, and we write $s \longrightarrow^* t$ for the reflexive and transitive closure of reduction.

A value of function type is a lambda expression, a constant, or a cast. If a constant is of function type, its meaning is specified by the function $[\![c]\!]$. For example, $+$ is a constant of type $\textbf{Int} \to \textbf{Int} \to \textbf{Int}$, with $[\![+]\!](3) = +_3$, where $+_3$ is a constant of type $\textbf{Int} \to \textbf{Int}$ and $[\![+_3]\!](4) = 7$. We assume that the meaning of constants is consistent with their type: if $\mathrm{ty}(c) = S \to T$ and value v has type S then $[\![c]\!](v)$ has type T.

The rules for applying a lambda expression and a constant are standard. The rule for applying a cast is as follows.

$$\begin{array}{l} (\langle S' \to T' \Leftarrow S \to T \rangle^{pn}\, v)\, w \\ \longrightarrow \\ \langle T' \Leftarrow T \rangle^{pn}\, (v\, (\langle S \Leftarrow S' \rangle^{np}\, w)) \end{array}$$

Here $v : S \to T$ and $w : S'$, and the whole term has type T'. The cast is broken into two smaller casts, each in opposite directions, reversing the blame labels on the argument cast.

A value of subset type is a labelled constant. The rule for casting a subset type is as follows.

$$\begin{array}{l} \langle T \Leftarrow S \rangle^{pn}\, c_S \\ \longrightarrow \\ \quad \textbf{if}\, t[x := c_B]\, \textbf{then}\, c_T\, \textbf{else blame}\, p \\ \text{if } S = \{x : B \mid s\}, T = \{x : B \mid t\} \end{array}$$

The cast reduces to a conditional that tests the appropriate predicate and returns the constant if the predicate is true, or fails (blaming the cast) if the predicate is false. The predicate may also fail to terminate or cast blame, in which case the conditional will do the same. On the left-hand side, the constant c is labelled with the subset type S, and on the right-hand side it is relabelled with the base type B to give it the right type in the predicate, and with label T if the predicate evaluates to true. If the predicate does not evaluate to true, the right-hand side contains a subterm, c_T, that is ill-typed; therefore there is a special typing rule (discussed below) which assigns a type to the right-hand side as a whole.

A value of dynamic type is a cast from a source type, which is deconstructed by casting back to a target type. If the source and target types are compatible, the two casts collapse to a single cast.

$$\begin{array}{l} \langle T \Leftarrow \textbf{Dyn} \rangle^{p'n'}\, (\langle \textbf{Dyn} \Leftarrow S \rangle^{pn}\, v) \\ \longrightarrow \\ \quad \langle T \Leftarrow S \rangle^{p'n}\, v, \\ \text{if } S \sim T \end{array}$$

The cast from S to **Dyn** makes the type less precise, and so should never assign positive blame, and the cast from **Dyn** to T makes the type more precise, and so should never assign negative blame, as discussed in Section 2.3. Hence two of the blame labels can safely be discarded, and the remaining two carry over to the new cast. This is further discussed in Section 4.

If the source and target types are not compatible, then the term fails, assigning positive blame to the cast to the target.

$$\begin{array}{l} \langle T \Leftarrow \textbf{Dyn} \rangle^{p'n'}\, (\langle \textbf{Dyn} \Leftarrow S \rangle^{pn}\, v) \\ \longrightarrow \\ \quad \textbf{blame}\, p', \\ \text{if } S \not\sim T \end{array}$$

Negative blame is only assigned to function arguments. Since the cast from the source is making the type less precise it should not be assigned positive blame, hence the blame must go to the cast to the target. This rule differs slightly from the rules used for gradual typing (Siek and Taha 2006) or in Sage (Gronski et al. 2006), in that it fails immediately if the types are not compatible. In those other systems, incompatibility of function argument or range types is discovered only if and when the function is applied. This difference fell out naturally from our formulation; it is not clear whether it is important, but it may have advantages in terms of catching errors earlier.

The last two reduction rules give the compatible closure of reduction, and ensure that computation fails immediately if a blame term becomes the locus of reduction.

3.3 Run-time type rules

Figure 3 presents additional rules for typing terms at run-time, and rules for implication and consistent substitution.

The two additional type rules ensure that the reduction of a cast to a subset type remains well typed. How the rules work is discussed in detail in the proof of the preservation property in Section 3.5. Although the rules look *ad hoc* and special purpose, they are actually special cases of similar rules for implication and subset types found in Gronski et al. (2006) and Ou et al. (2004).

The run-time type rules use an undecidable judgement that determines when one predicate implies another; $\Gamma \models s \Rightarrow t$ holds if whenever term $s \longrightarrow^* \textbf{true}$ then term $t \longrightarrow^* \textbf{true}$. The definition quantifies over all substitutions consistent with a given environment, which are identified by a second undecidable judgement; $\Gamma \models \sigma$ holds if σ is a substitution that maps variable x to a value of type T for every pair $x : T$ in Γ. Both of these judgements are taken directly from (Flanagan 2006), and they are the source of undecidability in his type system and in our run-time type system. A similar entailment judgement is used by (Ou et al. 2004).

Hence decidability, Proposition 2, holds only for the type rules of Figure 1, and fails when these are extended with the rules of Figure 3. However, it is easy to check that unicity, Proposition 1,

holds for the type rules in Figure 1 even when extended by those of Figure 3.

The good news is that undecidability is not a show stopper. We introduce the undecidable type rules precisely in order to prove preservation and progress. It is straightforward to decide whether a source term is typed, and straightforward to perform successive reduction steps on a term. Progress guarantees that we can perform these steps without getting stuck, preservation guarantees that the resulting terms are well typed. There is never a need to decide whether a term satisfies the undecidable rules, since this is guaranteed by preservation and progress!

3.4 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 four subtyping judgements: ordinary, positive, negative, and naive.

The ordinary subtyping rules are similar to those found in Flanagan (2006) and Ou et al. (2004). We write $\Gamma \vdash S <: T$ if S is a subtype of T in environment Γ. Function subtyping is contravariant in the domain and covariant in the argument. One subset type is a subtype of another if the predicate of the first implies the predicate of the second; this is determined using the implication judgement defined in Figure 3. This means that subtyping is undecidable, but this is not a hindrance, since our type system does not depend on subtyping. Defining subtyping as undecidable is natural, and means we can show more types are in the subtype relation, making our results more powerful.

However, contrary to what one may expect, no type is a subtype of **Dyn** other than **Dyn** itself. This differs from the rule of Gronski et al. (2006), which takes every type to be a subtype of **Dyn**. In our case, we only take S to be a subtype of T if a cast from S to T can never receive any blame. However, we have seen that a cast that makes types less precise, such as a cast from T to **Dyn**, may receive negative blame (but not positive blame); therefore it is not appropriate to take T as a subtype of **Dyn** (but it is appropriate to take T as a positive subtype of **Dyn**, as discussed below). The issues are very similar to the treatment of the contract **Any**, as discussed by Findler and Blume (2006).

In order to capture the situations in which positive and negative blame cannot occur, we factor the notion of subtype into two subsidiary relations, positive subtyping, written $\Gamma \vdash S <:^+ T$ and negative subtyping, written $\Gamma \vdash S <:^- T$. Intuitively, these judgements are defined to track the swapping of positive and negative blame labels that occurs with function types. The two judgements are defined in terms of each other, with the contravariant position in the function typing rule reversing the roles. We have $S <:^+ \mathbf{Dyn}$ and $\mathbf{Dyn} <:^- T$ for every type S and T. The intuitive reason for this is that casting to **Dyn** can never give rise to positive blame, and casting from **Dyn** can never give rise to negative blame. We only check inclusion between subtypes for positive subtyping: we have $S <:^+ T$ for subset types only when the predicate of the first implies the predicate of the second, but we have $S <:^- T$ for all subset types. The intuitive reason for this is that a failed cast to a subset type can only give rise to positive blame, not negative blame.

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 is covariant rather than contravariant for function arguments. Thus, we have $\mathbf{Int} \to \mathbf{Nat} <: \mathbf{Nat} \to \mathbf{Int}$ but $\mathbf{Nat} \to \mathbf{Nat} <:_n \mathbf{Int} \to \mathbf{Int}$. The formal concept $S <:_n T$ corresponds to the informal concept of S being more precise than T. In Section 4 we show that $S <:_n T$ if and only if $S <:^+ T$ and $T <:^- S$. (Note the reversal! Here we write $T <:^- S$ where above we had $S <:^- T$.) 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.

As examples, note we have the following:

$$\begin{array}{rcl} \mathbf{Int} \to \mathbf{Nat} & <: & \mathbf{Nat} \to \mathbf{Int} \\ \mathbf{Int} \to \mathbf{Nat} & <:^+ & \mathbf{Nat} \to \mathbf{Int} \\ \mathbf{Int} \to \mathbf{Nat} & <:^- & \mathbf{Nat} \to \mathbf{Int} \end{array}$$

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

The first line shows that subtyping is contravariant in its function argument, and the fourth lines shows that naive subtyping is covariant. The first line is equivalent to the second and third, and the fourth line is equivalent to the fifth and sixth.

3.5 Type safety

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

Lemma 3. *(Substitution) If $\Gamma \vdash s : S$ and $\Gamma,\ x : S \vdash t : T$, then $\Gamma \vdash t[x := s] : T$.*

Lemma 4. *(Canonical forms) Let v be a value that is well-typed in the empty context. One of three cases applies.*

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

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

Proof. By case analysis over reductions. We consider only the unusual cases.

- Consider the reduction
$$\begin{array}{l} \langle T \Leftarrow S \rangle^{pn}\ c_S \\ \longrightarrow \\ \quad \mathbf{if}\ t[x := c_B]\ \mathbf{then}\ c_T\ \mathbf{else\ blame}\ p, \\ \text{if } S = \{x : B \mid s\}, T = \{x : B \mid t\} \end{array}$$
This preserves types because of the run-time typing rule for conditionals.
- Consider the reduction
$$\mathbf{if\ true\ then}\ c_T\ \mathbf{else\ blame}\ p \longrightarrow c_T$$
If $T = B$ then this is well typed in the usual way. If $T = \{x : B \mid t\}$ with $t \neq \mathbf{true}$, then the left-hand side can only be well-typed by the run-time typing rule for conditionals,
$$\frac{\Gamma \vdash s : \mathbf{Bool} \qquad T = \{x : B \mid t\} \qquad \mathrm{ty}(c) = B \qquad \Gamma \vdash T\ \mathbf{wf} \qquad \Gamma \models s \Rightarrow t[x := c_B]}{\Gamma \vdash (\mathbf{if}\ s\ \mathbf{then}\ c_T\ \mathbf{else\ blame}\ p) : T,}$$
and we must have $s = \mathbf{true}$, in which case the right-hand side is well typed by the run-time type rule for constants
$$\frac{T = \{x : B \mid t\} \qquad \mathrm{ty}(c) = B \qquad \Gamma \vdash T\ \mathbf{wf} \qquad \Gamma \models \mathbf{true} \Rightarrow t[x := c_B]}{\Gamma \vdash c_T : T.}$$

- Consider the reduction

$$\textbf{if false then } c_T \textbf{ else blame } p \longrightarrow \textbf{blame } p$$

This does not match the hypothesis, because **blame** p is not a term. □

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

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

Proof. By induction on the structure of the typing derivation. We consider only the unusual cases.

- Consider the rule

$$\frac{T = \{x : B \mid t\} \qquad \mathrm{ty}(c) = B \qquad \Gamma \vdash T \textbf{ wf} \qquad \Gamma \models \textbf{true} \Rightarrow t[x := c_B]}{\Gamma \vdash c_T : T}$$

In this case, the typed term is a value.
- Consider the rule

$$\frac{\Gamma \vdash s : \textbf{Bool} \qquad T = \{x : B \mid t\} \qquad \mathrm{ty}(c) = B \qquad \Gamma \vdash T \textbf{ wf} \qquad \Gamma \models s \Rightarrow t[x := c_B]}{\Gamma \vdash (\textbf{if } s \textbf{ then } c_T \textbf{ else blame } p) : T}$$

Since $\Gamma \vdash s : \textbf{Bool}$, 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 c_T or **blame** p.
  - $s \longrightarrow s'$ for some s', and the term reduces to

$$\textbf{if } s' \textbf{ then } c_T \textbf{ else blame } p.$$

  - $s \longrightarrow$ **blame** p', for some p', and the term reduces to **blame** p'. □

In this case, simply knowing preservation and progress does not guarantee a great deal, since it does not rule out reduction to a blame term. However, Section 4 gives results that let us identify circumstances where certain kinds of blame cannot arise.

3.6 Typed and untyped lambda calculus

We introduce a separate grammar for untyped lambda calculus, and show how to map this into our typed lambda calculus. 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 $\lfloor t \rfloor$ is used to embed a typed term into an untyped term; it may be applied only if the typed term has type **Dyn**.

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.

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

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

There is an simple mapping that takes untyped terms into typed terms.

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

Every well-formed untyped term maps to a typed term with type **Dyn**.

Lemma 7. *We have* $\Gamma \vdash M$ **wf** *if and only if* $\Gamma \vdash \lceil M \rceil : \textbf{Dyn}$.

4. The Blame Theorem

Subtyping factors into positive and negative subtyping.

Lemma 8. *We have* $\Gamma \vdash S <: T$ *if and only if* $\Gamma \vdash S <:^+ T$ *and* $\Gamma \vdash S <:^- T$.

Proof. By induction on the derivation of the judgement.

- **Dyn** $<:$ **Dyn**
iff (by definition)
$S <:^+$ **Dyn** and **Dyn** $<:^- T$, with $S = T =$ **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 (by definition)
$S \to T <:^+ S' \to T'$ and $S \to T <:^- S' \to T'$.
- $\{x : B \mid s\} <: \{x : B \mid t\}$
iff (by definition)
$\forall x : B.\, s \Rightarrow t$
iff (by definition)
$\{x : B \mid s\} <:^+ \{x : B \mid t\}$ and $\{x : B \mid s\} <:^- \{x : B \mid t\}$.

In the reverse direction there are nine cases to consider, but the six not listed above (e.g., **Dyn** $<:^+ S \to T$ and $S \to T <:^-$ **Dyn**) trivially validate the implication. □

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.

Lemma 9. *We have* $\Gamma \vdash S <:_n T$ *if and only if* $\Gamma \vdash S <:^+ T$ *and* $\Gamma \vdash T <:^- S$.

Proof. By induction on the derivation of the judgement.

- $S <:_n$ **Dyn**
iff (by definition)
$S <:^+$ **Dyn** and **Dyn** $<:^- T$ with $S = T$.
- $S \to T <:_n S' \to T'$
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 (by definition)
$S \to T <:^+ S' \to T'$ and $S' \to T' <:^- S \to T$.
- $\{x : B \mid s\} <:_n \{x : B \mid t\}$
iff (by definition)
$\forall x : B.\, s \Rightarrow t$
iff (by definition)
$\{x : B \mid s\} <:^+ \{x : B \mid t\}$ and $\{x : B \mid s\} <:^- \{x : B \mid t\}$.

Again, in the reverse direction there are nine cases, but the six not listed above are trivial. □

The following is the central result of this paper. Note that the subterms of a term include any term in a refinement 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 in positive position is a positive subtype, $\langle T \Leftarrow S \rangle^{pn}$ s has $S <:^+ T$, and*
- *every cast with label p in negative position is a negative subtype, $\langle T \Leftarrow S \rangle^{np}$ s has $S <:^- T$, and*

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

Proof. By case analysis on the reduction. For each reduction that contains a cast, we consider each label on the left-hand side, and show that if the cast with that label satisfies the induction hypothesis, so does every cast with that label on the right-hand side.

- Consider the reduction
$$\begin{array}{l} \langle S' \to T' \Leftarrow S \to T \rangle^{pn}\ f\ v \\ \longrightarrow \\ \langle T' \Leftarrow T \rangle^{pn}\ (f\ (\langle S \Leftarrow S' \rangle^{np}\ v)) \end{array}$$
The label p appears positively on the left. By hypothesis we have $S \to T <:^+ S' \to T'$ for the term on the left, whence definition of $<:^+$ we have $S' <:^- S$ and $T <:^+ T'$, so the hypothesis is maintained for the term on the right.
The label n appears negatively on the left. By hypothesis we have $S \to T <:^- S' \to T'$ for the term on the left, whence definition of $<:^-$ we have $S' <:^+ S$ and $T <:^- T'$, so the hypothesis is maintained for the term on the right. (These two cases are identical, save for swapping p with n and $<:^+$ with $<:^-$.)
- Consider the reduction
$$\begin{array}{l} \langle T \Leftarrow S \rangle^{pn}\ c_S \\ \longrightarrow \\ \quad \textbf{if}\ t[x := c_B]\ \textbf{then}\ c_T\ \textbf{else blame}\ p, \\ \text{if } S = \{x : B \mid s\}, T = \{x : B \mid t\} \end{array}$$
The label p appears positively on the left. By hypothesis we have $S <:^+ T$ for the term on the left, whence by definition of $<:^+$ we have $\Gamma,\ x : B \models s \Rightarrow t$. Since $c_S : S$ we know that $s[x := c_B] \longrightarrow^*$ **true**, and it follows that $t[x := c_B] \longrightarrow^*$ **true**, so the right hand side reduces to c_T and not to **blame** p.
The label n appears negatively on the left and does not appear on the right, so preserves the hypothesis trivially. More correctly, n may appear independently in t, but in that case the hypothesis carries over independently.
- Consider the reduction
$$\begin{array}{l} \langle T \Leftarrow \textbf{Dyn} \rangle^{p'n'}\ (\langle \textbf{Dyn} \Leftarrow S \rangle^{pn}\ v) \\ \longrightarrow \\ \langle T \Leftarrow S \rangle^{p'n}\ v, \text{if } S \sim T \end{array}$$
The positive label p appears in a cast from S to **Dyn**, and $S <:^+$ **Dyn** by definition; but p does not appear on the right hand side, and so this reduction preserves the necessary invariant trivially.
Similarly, the negative label n' appears in a cast from **Dyn** to T, and **Dyn** $<:^-$ T by definition; but n' does not appear on the right hand side, and so this reduction preserves the necessary invariant trivially.
The positive label p' appears in a cast from **Dyn** to T, and **Dyn** $<:^+$ T holds only if T is **Dyn**, in which case the cast on the right is $\langle$**Dyn** $\Leftarrow S \rangle^{p'n}$ for which $S <:^+$ **Dyn** holds trivially.
Similarly, the negative label n appears in a cast from S to **Dyn**, and $S <:^-$ **Dyn** holds only if S is **Dyn**, in which case the cast on the right is $\langle T \Leftarrow$ **Dyn** $\rangle^{p'n}$ for which **Dyn** $<:^-$ T' holds trivially.
- Consider the reduction
$$\begin{array}{l} \langle T \Leftarrow \textbf{Dyn} \rangle^{p'n'}\ (\langle \textbf{Dyn} \Leftarrow S \rangle^{pn}\ v) \\ \longrightarrow \\ \quad \textbf{blame}\ p', \\ \text{if } S \not\sim T \end{array}$$
Of the four subcases for the previous reduction, the argument for the first two cases is the same here. In the remaining two cases, the hypothesis is satisfied only if S or T is **Dyn**, but that cannot happen since we have $S \not\sim T$, and **Dyn** is compatible with all types. □

We have an immediate corollary.

Corollary 11. *(Well-typed programs can't be blamed) Let t be a well-typed term with a subterm*
$$\langle T \Leftarrow S \rangle^{pn}\ s$$
containing the only occurrences of p and n in t.

- *If $S <:^+ T$ then $t \not\longrightarrow^*$ **blame** p.*
- *If $S <:^- T$ then $t \not\longrightarrow^*$ **blame** n.*
- *If $S <: T$ then $t \not\longrightarrow^*$ **blame** p and $t \not\longrightarrow^*$ **blame** n*

In particular, since $S <:^+$ **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 **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 $\langle T \Leftarrow S \rangle^{pn}\ s$ 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, regardless of how it was derived, and 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) define cross-language casts between Scheme and ML.

If e_S is a term in Scheme, they write

$$^{\tau}MS_N\ (\mathcal{G}^{\tau}_{+}\ e_S)$$

for the corresponding term of type τ in ML. Here $^{\tau}MS_N$ indicates conversion from Scheme to an ML term of type τ with no checking (hence the subscript N), while $\mathcal{G}^{\tau}_{+}$ is a "guard" in Scheme that ensures the term e_S satisfies the contract specified by τ, checking only for instances of positive blame (hence the subscript $+$). The corresponding conversion in our calculus is

$$\langle T \Leftarrow \mathbf{Dyn}\rangle^{pn}\ s$$

where T corresponds to τ and s corresponds to e_S. Since $\mathbf{Dyn} <:^{-} T$ 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_M is a term of type τ in ML, they write

$$\mathcal{G}^{\tau}_{-}\ (SM^{\tau}_{N}\ e_M)$$

for the corresponding term in Scheme. Here SM^{τ}_{N} indicates conversion from an ML term of type τ to a Scheme term with no checking (hence the subscript N), while $\mathcal{G}^{\tau}_{-}$ is a "guard" in Scheme that ensures the term e_M satisfies the contract specified by τ, checking only for instances of negative blame (hence the subscript $-$). The corresponding conversion in our calculus is

$$\langle \mathbf{Dyn} \Leftarrow T\rangle^{pn}\ t$$

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

5.4 Gronski and Flanagan

Gronski and Flanagan (2007) relate the contract calculus of Findler and Felleisen (2002), modeled as a calculus λ^C, to the hybrid type calculus of Flanagan (2006), modeled as a calculus λ^H.

The calculus λ^H of Gronski and Flanagan is similar to the calculus of this paper, but one key difference is that cast terms have only a single blame label.

$$\begin{array}{lllll} \text{Type} & S & ::= & \{x : B \mid s\} \mid S_1 \to S_2 \\ \text{Term} & s & ::= & \cdots \mid \langle S_2 \Leftarrow S_1\rangle^{l}\ s \end{array}$$

They also introduce a calculus λ^C to model the calculus λ^{Con} of Findler and Felleisen, that separates types from contracts.

$$\begin{array}{lllll} \text{Type} & T & ::= & B \mid T_1 \to T_2 \\ \text{Contract} & c & ::= & \mathbf{contract}\ B\ v \mid c_1 \mapsto c_2 \\ \text{Term} & t & ::= & \cdots \mid t^{c,l,l'} \end{array}$$

Here a contract on a base type $\mathbf{contract}\ B\ v$ takes a value v which must be a predicate of type $B \to \mathbf{Bool}$, and a contract on a function type $c_1 \mapsto c_2$ checks that its argument satisfies contract c_1 and its results satisfies contract c_2. A term t may be annotated with a contract c to be checked and labels l and l' for positive and negative blame respectively. Whereas in λ^H casts change types (as in our work), in λ^C contracts preserve types: if term t has type T, then the annotated term $t^{c,l,l'}$ also has type T.

They define a mapping ϕ from the types and terms of λ^C to those of λ^H, and show that it preserves types and reductions. The key to the mapping is that a term which enforces a contract maps to a *pair* of casts.

$$\phi(t^{c,l,l'}) \quad = \quad \langle \text{base}(S) \Leftarrow S\rangle^{l'}\ (\langle S \Leftarrow \text{base}(S)\rangle^{l}\ \phi(t)) \quad \text{where } S = \phi_c(c)$$

Here ϕ_c maps a contract of λ^C into a subset type of λ^H, and base erases a subset type of λ^H to obtain a simple type of λ^C.

$$\begin{array}{lcl} \phi_c(\mathbf{contract}\ B\ v) & = & \{x : B \mid \phi(v)\ x\} \\ \phi_c(c_1 \mapsto c_2) & = & \phi_c(c_1) \to \phi_c(c_2) \\ \text{base}(\{x : B \mid s\}) & = & \{x : B \mid \mathbf{true}\} \\ \text{base}(S_1 \to S_2) & = & \text{base}(S_1) \to \text{base}(S_2) \end{array}$$

This result is easily understood in terms of our results, since $\text{base}(S) <:^{-} S$, so the rightmost cast can only allocate positive blame, and $S <:^{+} \text{base}(S)$, so the leftmost cast can only allocate negative blame.

However, in general λ^H contains casts of the form $\langle S_2 \Leftarrow S_1\rangle^{l} s$ where neither $S_1 <:^{-} S_2$ nor $S_1 <:^{+} S_2$ holds, so we would argue that their simplicity is misguided: 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 (perhaps counterintuitive) observation that one should not regard all types as subtypes of the type 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 perhaps simpler way.

Gray et al. (2005) discuss the practice of using contract 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. This 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 this paper apply only to simple types and type Dynamic, and it is not yet clear how to extend them to the subset types found in hybrid type systems.

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. The theory of Sage extends hybrid types in that it adds a type `Dynamic`, similar to our **Dyn**. It also supports first-class types, and permits the subtype construction to be applied to any types (not just base 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 here of the notation $\lceil M \rceil$ to embed untyped lambda calculus into our typed calculus.

7. Conclusion

Organisms evolve by selection of the fittest, and programming languages evolve in the same way, but fittest may not mean best! Gould (1994) repeatedly remarks on the fact that evolution leads to creatures which reproduce well in a given environment, not necessarily to creatures which are more sophisticated. In the developer environment, anecdotal evidence suggests that familiarity, libraries, development tools, user community, or other network effects may drown out actual technical superiority when comparing languages.

Can we promote the evolution of programming languages by some mechanism other than 'survival of the fittest'? Integrating two competing designs into a single language may provide a better basis for comparing the strengths and weaknesses of each, factoring out the biases listed above.

Acknowledgements

This paper benefited enormously from conversations with John Hughes. We also thank Matthias Felleisen, Cormac Flanagan, Oleg Kisselyov, 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. URL `citeseer.ist.psu.edu/abadi89dynamic.html`.

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.

Stephen Jay Gould. *Eight Little Piggies*. Penguin, 1994.

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 *Scheme and Functional Programming Workshop (Scheme)*, September 2006.

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.

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 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 *Scheme and Functional Programming Workshop (Scheme)*, 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. URL `http://portal.acm.org/citation.cfm?id=646242.681286`.

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

Andrew K. Wright and Robert Cartwright. A practical soft typing system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997. URL `http://portal.acm.org/citation.cfm?id=239912.239917`.

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.