Genericity, Nominal Inheritance and Gradual Typing

Jukka Lehtosalo and David J. Greaves

University of Cambridge Computer Laboratory
firstname.lastname@cl.cam.ac.uk

Abstract. A gradual type system allows evolving a program from dynamic typing into static typing smoothly, as intermediate versions may freely mix static and dynamic typing. Dynamically-typed programs tend to make heavy use of heterogeneous container objects, and generic types enable type-safe modelling of such objects. We present a gradual type system for an object-oriented language that supports generic types. The language allows a single generic instance to be coerced to multiple types with different typing precision in parameter values, in order to support flexible sharing of instances between dynamically-typed and statically-typed code. Our type system also allows mixing static and dynamic typing in nominal inheritance and overriding. Together these features allow flexible interaction between dynamically-typed and statically-typed parts of a program. We prove that runtime type errors are always caused by the dynamically-typed portion of the program, and we assign blame properly.

1 Introduction

A dynamically-typed scripting language such as Python or Ruby allows a team to quickly develop the initial releases of a program. Previous studies, though inconclusive, have reported over five-fold productivity differences between scripting languages and more traditional languages such as Java for some tasks [22]. As time passes, the program – and often also the development organisation – become larger and more complex. Eventually dynamic typing becomes a burden: the program gets difficult to modify, debug and understand. Static typing would allow a smoother maintenance phase, but at the expense of less rapid initial development.

Gradual transformation of dynamically-typed (or 'untyped') programs into statically-typed (or 'typed') programs is a potential solution for combining benefits of static and dynamic typing [25, 24, 19]. This paper presents a gradual type system for a Java-like language that makes two main contributions towards reaching the above goal: it supports gradual typing for generic types and nominal inheritance.

A practical gradual type system must support flexible mixing of untyped and typed code. Scripting languages rely heavily on libraries for improving productivity, and these libraries should be mainly statically-typed to allow type-safe access from typed programs. A gradual type system should therefore allow untyped programs to access typed libraries. In particular, untyped classes must be able to override typed methods defined in library classes, and collections such as lists with untyped items must be compatible with precisely typed collections.

Flexible collection objects such as lists and hash tables are ubiquitous in Python or Ruby code. Without type system support for collection types, reading from a container would require explicit casts (as in pre-1.5 Java), resulting in the addition of numerous unsafe casts when adapting untyped code. Alternatively, some languages support implicit downcasts, but this sacrifices type safety without a clear syntactic indication of this in the program. Our language supports generic types for providing type-safe collection types.

Statically-typed languages generally support generic types in one of two possible ways. The first option uses *type erasure* [7] (as in Java generics), where generic type parameters are only used during type checking and compilation, and they are erased before evaluation. In *reification*, generic instances embed runtime representations of the type variable values (as in Java arrays and .NET generics [20]). In this paper we show that neither implementation technique, as such, is a good fit for our gradual type system.

We present a gradual type system with generic types that supports coercions between generic types with different type arguments. For example, an instance of type `List<dyn>` (an untyped list) can be coerced into a value of type `List<String>`. Accessing the list object with the latter type may generate runtime type errors (if the list contains objects other than strings); we detect these errors and blame the coercion from `List<dyn>` to `List<String>`. We call references to an object with different, potentially incompatible types, such as `List<dyn>` and `List<String>`, different *views* of an object. This approach allows flow-sensitive use of generic type instances, with different views in different parts of a program. The same instance can be used as untyped in an untyped part of a program and as typed in other parts.

Our language also supports gradual typing for nominal *mixed inheritance* by allowing an untyped method to override a typed one. More generally, a less precisely typed method may override a more precisely typed one. We implement mixed inheritance by automatically generating multiple variants of a method, one for each different signature in the class hierarchy.

We use wrapper objects to implement coercions between generic types. For example, a coercion from `List<String>` to type `List<dyn>` results in a wrapped list object that checks that types of incoming values have type `String` at runtime. Only one wrapper is kept for a single instance reference at a time; multiple wrappers are combined to a wrapper that is at least as specific as each separately. For example, wrappers for `Map<Int, dyn>` and `Map<Object, String>` combine into a `Map<Int, String>` wrapper.

Our implementation technique also supports *naked* unwrapped references that do not require coercions when accessed. As an object is constructed, the result is a naked object; wrappers are added only if the value is coerced to a type with different level of static typing precision. Thus fully typed programs require no wrappers.

These are the main contributions of this paper:

1. We present a gradual type system for an object-oriented language with nominal subtyping which supports flexible mixing of dynamic and static types in implementation inheritance and method overriding. Our language also supports mixed generic inheritance with overriding: an untyped class can extend a generic, statically-typed class.
2. We present a language that allows arbitrary mixing of typed and untyped generics via multiple runtime views to a generic instance. We also extend runtime views to enable covariant and contravariant coercions between generic types with different type arguments, e.g. between `A<X>` and `A<Y>` when `X` is a subtype of `Y`. Our technique preserves runtime type safety without requiring any variance declarations [18]. This technique can also be useful for traditional typed object-oriented languages without gradual typing.

3. We formalise a core language that includes gradual typing, mixed inheritance, generics and blame and we prove the type safety of the language. We also prove that all runtime errors can be blamed on an imprecisely-typed part of the program.
4. We present space- and time-efficient implementation techniques for the gradual type system via program transformation. No runtime wrapper objects are needed for purely typed or untyped code or for mixed inheritance. No runtime type checks are needed when accessing generic instances that are only used in a statically-typed section of program (unlike Java generics).

The paper is structured as follows. In Section 2 we give an overview of gradual typing along with the language and the type system, using examples. We show how our approach supports several useful programming idioms that are specific to gradual typing. In Section 3 we formally specify the language and prove soundness and other properties. The language is FJ extended with the dynamic type, generic types and blame. Section 4 introduces techniques for implementing the language efficiently. We discuss related work in Section 5 and conclude in Section 6.

2 Motivation: Why gradual typing?

This section presents several examples as motivation for using a gradual type system. In the simplest case, an untyped program may use a typed library. Here we use a typed widget library from untyped code (untyped code is within `|...|`); dynamically-typed values have type `dyn`):

```
dyn parent = ...;
dyn button = |new Button(parent, "Title")|;
```

A gradual type system verifies that the constructor is called with compatible argument types. If the `parent` argument has an invalid type, a pure untyped language such as Python[1] might silently accept the argument during `Button` construction and only raise an exception later, such as when clicking the button, which makes it difficult to find the ultimate reason for the error during debugging.

If a typed library expects a generic instance such as a list, checking the type efficiently at runtime is not straightforward. In this example we assume that `ListBox` expects a string array, but it is given a mixed array with a string and a number:

```
dyn list = |new ListBox(parent, new [] { "item", 2.0 } )|;
```

Our approach is to check the list items lazily when accessing them in the `ListBox` class. This generates an exception somewhere within the `ListBox` implementation, but this can happen later after the constructor has finished. To facilitate debugging, we remember that we bound the untyped list to a typed list in the above call (and we could not be sure that this was safe). At runtime type error, we can blame this location as the source of the type error. In a complex library which deals with many generic instances received from the client, blame is important by helping locate errors quickly.

Untyped programs often extend library classes. For example, a program may extend a widget class defined in an user interface framework and override some methods.

[1] Or a language with an optional type system.

```
class MyWidget extends widgetlib.Widget {
  ...
  dyn getChildren() {
    dyn items = ...;
    return new Set(items);
  }
}
```

We assume that the typed `getChildren` method of `Widget` is declared to return a `List`, which is not compatible with `Set`:

```
package widgetlib;

public class Widget {
  ...
  List<Widget> getChildren() { ... }
  ...
}
```

A gradual type system should detect the error in `MyWidget` at runtime[2]. Assume that the actual type error happens within library code, when calling `getChildren` via a typed reference:

```
Widget widget = <reference to MyWidget>;
List<Widget> children = widget.getChildren();
```

Now we blame the `getChildren` method override in `MyWidget`. We can point to an error source in the untyped program, and the programmer does not have to debug the internals of the widget library.

In the above example, the `getChildren` method could also return the correct instance type `List<dyn>` but with incompatible item values. We would check list items lazily, again, and blame the override.

Previous examples had an untyped program and a typed library. Untyped modules can also be gradually adapted to static typing. Since a static type system cannot model all behaviour supported by dynamic typing, this is more complex than simply adding type annotations, at least for non-trivial programs – programs often require refactoring. Additionally, there are generally multiple self-consistent typing assignments for a program. Some of these may be incompatible with untyped code that accesses the module and, generally, this can be only checked by testing.

To simplify debugging, the evolution from dynamic to static typing is best done in small steps, running tests after each modification to verify that the last change preserved behaviour. This helps localise errors and reduces debugging time. A module or a class gradually gets more and more typing coverage, and we must be able to run the program at each intermediate position to check for runtime type errors.

An untyped module may contain structures that require major work to adapt to static typing, and this adaptation may also make the code more complex and difficult to maintain. A gradual type system allows the programmer to decide individually for each piece of code whether it is a good fit for static typing. It is always possible to leave pockets

[2] Of course, the error in this example could be detected by simple static analysis, but this does not generalise to more complex programs.

of untyped code in an otherwise typed module. Perheps the perceived cost in extra complexity is larger than the benefit of having static typing, or perhaps the typed version would still have many casts or other unsafe features, which would make static typing less desirable.

It takes non-trivial effort to adapt a large untyped codebase to static typing. A busy development team may find it difficult to commit itself to this work, as they will reap the benefits of reduced maintenance costs only in the long term. In a context like this, gradual typing allows adding static types selectively where they make the biggest difference: to complex algorithms, performance-critical sections and module interfaces (while keeping module implementations untyped). The development team starts getting benefits from static typing immediately, with very little development commitment.

Our techniques support all of the above cases. The next section discusses the language in more technical detail.

3 Formalisation and soundness

$\mathtt{C}, \mathtt{D}$	class name	$\mathtt{S}, \mathtt{T}, \mathtt{U}, \mathtt{V}, \mathtt{W}, \mathtt{R}, \mathtt{Q}$	type	$\mathtt{X}$	type variable
$\mathtt{v}, \mathtt{u}, \mathtt{w}$	value	$\mathtt{f}, \mathtt{g}$	field name	ℓ	label

$$
\begin{array}{rcll}
\mathtt{CL} & ::= & \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}\ \mathtt{extends}\ \mathtt{C}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \} & \text{class definition} \\
\mathtt{K} & ::= & \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{f}});\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} & \text{constructor} \\
\mathtt{M} & ::= & \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})^{\ell}\ \{\ \mathtt{return}\ \mathtt{e};\ \} & \text{method} \\
\mathtt{e} & ::= & \mathtt{x}\ \mid \mathtt{e}.\mathtt{f}^{\ell}\ \mid \mathtt{e}.\mathtt{m}(\overline{\mathtt{e}})^{\ell}\ \mid \mathtt{new}\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})^{\ell}\ \mid (\mathtt{T})^{\ell}\mathtt{e}\ \mid \lfloor \mathtt{e} \rfloor & \text{expression} \\
\mathtt{T} & ::= & \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ \mid \mathtt{X}\ \mid \mathtt{dyn} & \text{type}
\end{array}
$$

Fig. 1. Notation and syntax of the source language.

This section introduces a formalisation of a core subset of our language, which is an extension of Featherweight Java (FJ). In our presentation we highlight differences between our language and FGJ, an extension of FJ with genericity based on *type erasure* [17]. We omit a detailed discussion of some features common with FGJ to conserve space.

3.1 Notation

Our presentation mostly follows the conventions used in the original FJ and FGJ paper. Unlike FJ, we use a strict evaluation order, similar to Pierce [23]. Figure 1 explains some main notational conventions and the source language syntax. We use $\overline{\mathtt{C}}$ as shorthand for $\mathtt{C}_1,\ldots,\mathtt{C}_n$. We use $\bullet$ for empty list and $\#(\overline{\mathtt{L}})$ for list length. We use $[\overline{\mathtt{X}}/\overline{\mathtt{Y}}]\mathtt{e}$ for substituting $\mathtt{Y}_1 \rightarrow \mathtt{X}_1,\ \ldots,\mathtt{Y}_n \rightarrow \mathtt{X}_n$ in $\mathtt{e}$. Additional conventions include using $\overline{\mathtt{dyn}}$ as shorthand for the list $\mathtt{dyn},\ \ldots,\mathtt{dyn}$ of unspecified length and $\mathit{meet}(\overline{\mathtt{T}},\overline{\mathtt{S}})$ to signify the list $\mathit{meet}(\mathtt{T}_1,\mathtt{S}_1),\ \ldots,\mathit{meet}(\mathtt{T}_n,\mathtt{S}_n)$ (and similarly for other functions). We use $\mathit{ftv}(\mathtt{T})$ for the set of free type variables in $\mathtt{T}$.

3.2 Syntax

Our syntax includes the syntactic form $\lfloor e \rfloor$ for embedded untyped expressions. Unlike FJ, we also include labels (ℓ) in methods, new expressions, casts, method calls and field access to expressions to track blame. These labels would not be explicit in actual programs.

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{T} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{S}}{\mathtt{T} <: \mathtt{S}} \qquad \mathtt{X} <: \mathtt{Object} \qquad \frac{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\rangle\ \mathtt{extends}\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{\ldots\} \qquad \mathtt{D}\langle[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\rangle = \mathtt{D}\langle\overline{\mathtt{U}}\rangle}{\mathtt{C}\langle\overline{\mathtt{T}}\rangle <: \mathtt{D}\langle\overline{\mathtt{U}}\rangle}$$

$$\mathtt{T} \sim \mathtt{T} \qquad \mathtt{dyn} \sim \mathtt{T} \qquad \mathtt{T} \sim \mathtt{dyn} \qquad \frac{\overline{\mathtt{T}} \sim \overline{\mathtt{S}}}{\mathtt{C}\langle\overline{\mathtt{T}}\rangle \sim \mathtt{C}\langle\overline{\mathtt{S}}\rangle}$$

$$\frac{\mathtt{T} <: \mathtt{S}}{\mathtt{T} \lesssim \mathtt{S}} \qquad \frac{\mathtt{T} \sim \mathtt{S}}{\mathtt{T} \lesssim \mathtt{S}} \qquad \frac{\mathtt{T} \lesssim \mathtt{U} \qquad \mathtt{U} \lesssim \mathtt{S}}{\mathtt{T} \lesssim \mathtt{S}} \qquad \frac{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\rangle\ \mathtt{extends}\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{\ldots\} \qquad \mathtt{D}\langle[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\rangle \sim \mathtt{D}\langle\overline{\mathtt{U}}\rangle}{\mathtt{C}\langle\overline{\mathtt{T}}\rangle \lesssim \mathtt{D}\langle\overline{\mathtt{U}}\rangle}$$

$$\mathtt{T} \prec_{=} \mathtt{T} \qquad \mathtt{T} \prec_{=} \mathtt{dyn} \qquad \frac{\overline{\mathtt{T}} \prec_{=} \overline{\mathtt{S}}}{\mathtt{C}\langle\overline{\mathtt{T}}\rangle \prec_{=} \mathtt{C}\langle\overline{\mathtt{S}}\rangle}$$

Fig. 2. Subtyping ($<:$), consistency ($\sim$), subtype-or-consistent ($\lesssim$) and type precision ($\prec_{=}$).

3.3 Types, subtyping and consistency

Like previous work [25, 24, 30, 2, 19], our language includes a special type (`dyn`) for untyped values. `dyn` is compatible (or *consistent*) with every other type. We use $\sim$ for consistency, $<:$ for subtyping and $\lesssim$: for subtyping or consistency (Figure 2). Generic types are compatible even if the arguments are different but compatible. For example, `List<dyn>` is compatible with `List<String>`, but `List<String>` is not compatible with `List<Object>` (but these can be coerced into each other, as explained later). This is a main contribution of our work. It has a significant effect on evaluation. Additionally, we use $\prec_{=}$ for type precision relation: if $\mathtt{T} \prec_{=} \mathtt{S}$, we say that `T` is more precise than `S`. `List<String>` is more precise than both `List<dyn>` and `dyn`, but not vice versa.

3.4 Auxiliary functions

Several auxiliary functions are defined in Figure 3. Many of these are similar but generally simpler than corresponding functions in FGJ (*mtype*, *mbody*, *fields*). The differences stem from omitting bounded quantification and generic methods for clarity.

The *validOverride* function determines whether a method override has a valid signature. Unlike FJ and Ina and Igarashi [19], we allow *less-precise* singatures in overrides. For example, we support overriding signature `Int m(List<String> x)` with `dyn m(dyn x)` or `Int m(List<dyn> x)`. This is another of our main contributions; it significantly affects evaluation.

nfields looks up the number of fields in a class (including inherited fields) and *cargs* looks up the type arguments of a class.

Valid method override:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{S}}\texttt{> \{...\}} \quad mtype(\texttt{m}, [\overline{\texttt{V}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{S}}\texttt{>}) = \overline{\texttt{U}} \rightarrow \texttt{U}_0 \text{ implies } \overline{\texttt{U}} \prec_= \overline{\texttt{T}} \text{ and } \texttt{U}_0 \prec_= \texttt{T}_0}{validOverride(\texttt{C<}\overline{\texttt{V}}\texttt{>}, \texttt{m}, \overline{\texttt{T}} \rightarrow \texttt{T}_0)}$$

Method type lookup:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{V}}\texttt{> \{ }\overline{\texttt{U}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{ \}} \qquad \texttt{S}_0\ \texttt{m}(\overline{\texttt{S}}\ \overline{\texttt{x}})^{\ell}\ \texttt{\{ return e; \}} \in \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\overline{\texttt{S}} \rightarrow \texttt{S}_0)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{V}}\texttt{> \{ }\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mtype(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{V}}\texttt{>})}$$

Method body lookup:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{V}}\texttt{> \{ }\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{ \}} \qquad \texttt{U}_0\ \texttt{m}(\overline{\texttt{U}}\ \overline{\texttt{x}})^{\ell}\ \texttt{\{ return e; \}} \in \overline{\texttt{M}}}{mbody(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{x}}\,.\,[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{e}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{V}}\texttt{> \{ }\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mbody(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mbody(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{V}}\texttt{>})}$$

Field and type argument lookup:

$$fields(\texttt{Object}) = \bullet \qquad \frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{V}}\texttt{> \{ }\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{ \}} \qquad fields([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{V}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{fields(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}, \overline{\texttt{U}}\ \overline{\texttt{g}}}$$

$$\frac{cargs(\texttt{C}) = \overline{\texttt{X}} \qquad fields(\texttt{C<}\overline{\texttt{X}}\texttt{>}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{nfields(\texttt{C}) = \#(\overline{\texttt{f}})} \qquad \frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{S}}\texttt{> \{ }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{ \}}}{cargs(\texttt{C}) = \overline{\texttt{X}}}$$

Fig. 3. Auxiliary functions.

3.5 Typing

Our static typing rules are fairly standard, the main addition to FGJ being that we allow type consistency in addition to subtyping and we support *dynamic casts* to dyn (Figures 4 and 5). Other differences stem from the omission of bounded quantification and generic methods.

The programmer can mix typed and untyped code without explicit casts or coercions, making integration smooth, as was shown in the examples in the Section 2.

$$\frac{\texttt{X} \in \Delta}{\Delta \vdash \texttt{X}\ \text{OK}} \qquad \Delta \vdash \texttt{dyn}\ \text{OK} \qquad \frac{\Delta \vdash \overline{\texttt{T}}\ \text{OK} \qquad cargs(\texttt{C}) = \overline{\texttt{X}} \qquad \#(\overline{\texttt{T}}) = \#(\overline{\texttt{X}})}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{OK}}$$

Fig. 4. Well-formed types.

3.6 Coercion and guard insertion transformation

We transform the program to a target language with explicit coercions before evaluation. We insert coercions and *guards* that explicitly coerce between untyped and typed values and generally between values of two arbitrary types. Figure 6 contains the syntax and evaluation contexts for the target language. Figure 7 contains transformation rules for

Expression typing:

$$\textsc{T-Var}\quad \Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \textsc{T-Field}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{C<\overline{T}>} \qquad \mathit{fields}(\mathtt{C<\overline{T}>}) = \overline{\mathtt{S}}\,\overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e.f}_i^{\ell} : \mathtt{S}_i} \qquad \textsc{T-DyField}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{dyn}}{\Delta;\Gamma \vdash \mathtt{e.f}^{\ell} : \mathtt{dyn}}$$

$$\textsc{T-Invk}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{C<\overline{T}>} \qquad \mathit{mtype}(\mathtt{m},\mathtt{C<\overline{T}>}) = \overline{\mathtt{S}} \to \mathtt{S} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} \lesssim \overline{\mathtt{S}}}{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}})^{\ell} : \mathtt{S}} \qquad \textsc{T-DyInvk}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{dyn} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}})^{\ell} : \mathtt{dyn}}$$

$$\textsc{T-Creat}\quad \frac{\mathit{fields}(\mathtt{C<\overline{T}>}) = \overline{\mathtt{S}}\,\overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} \lesssim \overline{\mathtt{S}} \qquad \Delta \vdash \mathtt{C<\overline{T}>}\ \mathrm{OK}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C<\overline{T}>}(\overline{\mathtt{e}})^{\ell} : \mathtt{C<\overline{T}>}} \qquad \textsc{T-Cast}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{D<\overline{S}>} \qquad \Delta \vdash \mathtt{C<\overline{T}>}\ \mathrm{OK} \qquad \mathtt{D<\overline{S}>} <: \mathtt{C<\overline{T}>} \text{ or } \mathtt{C<\overline{T}>} <: \mathtt{D<\overline{S}>}}{\Delta;\Gamma \vdash (\mathtt{C<\overline{T}>})^{\ell}\mathtt{e} : \mathtt{C<\overline{T}>}}$$

$$\textsc{T-DyCast}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Delta;\Gamma \vdash (\mathtt{dyn})\mathtt{e} : \mathtt{dyn}} \qquad \textsc{T-DyCast2}\quad \frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{dyn} \qquad \Delta \vdash \mathtt{C<\overline{T}>}\ \mathrm{OK}}{\Delta;\Gamma \vdash (\mathtt{C<\overline{T}>})^{\ell}\mathtt{e} : \mathtt{C<\overline{T}>}}$$

Method typing:

$$\textsc{T-Method}\quad \frac{\Delta = \overline{\mathtt{X}} \qquad \Delta \vdash \overline{\mathtt{T}}\ \mathrm{OK} \qquad \Delta \vdash \mathtt{T}_0\ \mathrm{OK} \qquad \Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<\overline{X}>} \vdash \mathtt{e} : \mathtt{S}_0 \qquad \mathtt{S}_0 \lesssim \mathtt{T}_0 \qquad \mathit{validOverride}(\mathtt{C<\overline{X}>},\mathtt{m},\overline{\mathtt{T}} \to \mathtt{T}_0)}{\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\,\overline{\mathtt{x}})^{\ell}\ \{\ \mathtt{return\ e};\ \}\ \mathrm{OK\ IN}\ \mathtt{C<\overline{X}>}}$$

Class typing:

$$\textsc{T-Class}\quad \frac{\mathtt{K} = \mathtt{C}(\overline{\mathtt{S}}\,\overline{\mathtt{g}},\overline{\mathtt{T}}\,\overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \qquad \mathit{fields}(\mathtt{D<\overline{U}>}) = \overline{\mathtt{S}}\,\overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathrm{OK\ IN}\ \mathtt{C<\overline{X}>} \qquad \overline{\mathtt{X}} \vdash \overline{\mathtt{T}}\ \mathrm{OK} \qquad \overline{\mathtt{X}} \vdash \mathtt{D<\overline{U}>}\ \mathrm{OK}}{\mathtt{class}\ \mathtt{C<\overline{X}>}\ \mathtt{extends}\ \mathtt{D<\overline{U}>}\ \{\ \overline{\mathtt{T}}\,\overline{\mathtt{f}};\mathtt{K}\,\overline{\mathtt{M}}\ \}\ \mathrm{OK}}$$

Fig. 5. Typing for the source language.

expressions, methods and classes. For example, call to constructor $\mathtt{C<X>}(\mathtt{List<X>},\mathtt{dyn})$ in the following code would be transformed like this (rule TR-CREAT), assuming declarations `dyn d` and `D x`:

$$\mathtt{new\ \ C<B>}(\mathtt{d},\mathtt{x})^{\ell} \rightsquigarrow \langle\mathtt{C<B>}\rangle\mathtt{new\ \ C<B} \Leftarrow \mathtt{B>}_{\circ}(\langle\mathtt{List<B>} \Leftarrow_{\ell} \mathtt{dyn}\rangle\mathtt{d}, \langle\mathtt{dyn} \Leftarrow_{\ell} \mathtt{D}\rangle\mathtt{x})$$

Runtime values v keep track of labels; the empty label $\circ$ is used initially before any coercions when blame cannot occur. Coercions that may cause type errors at runtime have non-empty labels. We also keep track of the type variables of generic instances at runtime. Since we support coercions between different generic types, a value contains both the original type arguments, the greatest lower bounds of type arguments values and the current access type of the value. For example, a string list value can be of form

$$\langle\mathtt{Collection<Object>}\rangle\mathtt{new\ List<String} \Leftarrow \mathtt{dyn>}_{\ell}(\ldots).$$

The list was created as `List<dyn>` and it was later coerced to `List<String>`; it is currently accessed as `Collection<Object>` (as a result of another set of coercions). We do not maintain a detailed history of all coercions. To avoid arbitrary runtime space

$$
\begin{array}{lll}
\mathsf{e} ::= \mathsf{x} \mid \mathsf{e.f}^{\ell} \mid \langle \mathsf{e.m}(\overline{\mathsf{e}})\rangle^{\ell} \mid \mathsf{v.m}(\overline{\mathsf{e}}) \mid \langle \mathsf{T}\rangle \mathsf{new}\ \mathsf{C}{<}\overline{\mathsf{T}} \Leftarrow \overline{\mathsf{T}}{>}_{\ell}(\overline{\mathsf{e}}) \mid (\mathsf{T})^{\ell}\mathsf{e} & \text{expression} \\
\quad \mid \langle \mathsf{T} \Leftarrow_{\ell} \mathsf{T}\rangle \mathsf{e} \mid \mathsf{blame}\ \ell & \\
\mathsf{v} ::= \langle \mathsf{T}\rangle \mathsf{new}\ \mathsf{C}{<}\overline{\mathsf{T}} \Leftarrow \overline{\mathsf{T}}{>}_{\ell}(\overline{\mathsf{v}}) & \text{value} \\
 & \\
E ::= [\,] \mid E.\mathsf{f}^{\ell} \mid \langle E.\mathsf{m}(\overline{\mathsf{e}})\rangle^{\ell} \mid \langle \mathsf{v.m}(\overline{\mathsf{v}}, E, \overline{\mathsf{e}})\rangle^{\ell} \mid \mathsf{v.m}(\overline{\mathsf{v}}, E, \overline{\mathsf{e}})^{\ell} & \text{evaluation context} \\
\quad \mid \langle \mathsf{U}\rangle \mathsf{new}\ \mathsf{C}{<}\overline{\mathsf{T}} \Leftarrow \overline{\mathsf{S}}{>}_{\ell}(\overline{\mathsf{v}}, E, \overline{\mathsf{e}}) \mid (\mathsf{T})^{\ell}E \mid \langle \mathsf{T} \Leftarrow_{\ell} \mathsf{S}\rangle E &
\end{array}
$$

Fig. 6. Syntax and evaluation contexts for the target language (differences from source language).

Expression transformation:

$$
\text{TR-INVK}\quad
\frac{\Delta;\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathsf{C}{<}\overline{\mathsf{U}}{>} \qquad \Delta;\Gamma \vdash \overline{\mathsf{e}} \rightsquigarrow \overline{\mathsf{e}}' : \overline{\mathsf{T}} \qquad \mathit{mtype}(\mathsf{m}, \mathsf{C}{<}\overline{\mathsf{U}}{>}) = \overline{\mathsf{S}} \to \mathsf{S}_0}{\Delta;\Gamma \vdash \mathsf{e.m}(\overline{\mathsf{e}})^{\ell} \rightsquigarrow \langle \mathsf{e}'.\mathsf{m}(\langle \overline{\mathsf{S}} \Leftarrow_{\ell} \overline{\mathsf{T}}\rangle \overline{\mathsf{e}}')\rangle^{\ell} : \mathsf{S}_0}
$$

$$
\text{TR-DYINVK}\quad
\frac{\Delta;\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathsf{dyn} \qquad \Delta;\Gamma \vdash \overline{\mathsf{e}} \rightsquigarrow \overline{\mathsf{e}}' : \overline{\mathsf{T}} \qquad \overline{\mathsf{S}} = \overline{\mathsf{dyn}} \qquad \#(\overline{\mathsf{S}}) = \#(\overline{\mathsf{T}})}{\Delta;\Gamma \vdash \mathsf{e.m}(\overline{\mathsf{e}})^{\ell} \rightsquigarrow \langle \mathsf{e}'.\mathsf{m}(\langle \overline{\mathsf{S}} \Leftarrow_{\circ} \overline{\mathsf{T}}\rangle \overline{\mathsf{e}}')\rangle^{\ell} : \mathsf{dyn}}
$$

$$
\text{TR-CREAT}\quad
\frac{\mathit{fields}(\mathsf{C}{<}\overline{\mathsf{U}}{>}) = \overline{\mathsf{S}}\ \overline{\mathsf{f}} \qquad \Delta;\Gamma \vdash \overline{\mathsf{e}} \rightsquigarrow \overline{\mathsf{e}}' : \overline{\mathsf{T}}}{\Delta;\Gamma \vdash \mathsf{new}\ \mathsf{C}{<}\overline{\mathsf{U}}{>}(\overline{\mathsf{e}})^{\ell} \rightsquigarrow \langle \mathsf{C}{<}\overline{\mathsf{U}}{>}\rangle \mathsf{new}\ \mathsf{C}{<}\overline{\mathsf{U}} \Leftarrow \overline{\mathsf{U}}{>}_{\circ}(\langle \overline{\mathsf{S}} \Leftarrow_{\ell} \overline{\mathsf{T}}\rangle \overline{\mathsf{e}}') : \mathsf{C}{<}\overline{\mathsf{U}}{>}}
\qquad
\text{TR-CAST}\quad
\frac{\Delta;\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathsf{T}}{\Delta;\Gamma \vdash (\mathsf{S})^{\ell}\mathsf{e} \rightsquigarrow (\mathsf{S})^{\ell}\mathsf{e}' : \mathsf{S}}
$$

$$
\text{TR-FIELD}\quad
\frac{\Delta;\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathsf{T} \qquad \Delta;\Gamma \vdash \mathsf{e.f}^{\ell} : \mathsf{S}}{\Delta;\Gamma \vdash \mathsf{e.f}^{\ell} \rightsquigarrow \mathsf{e}'.\mathsf{f}^{\ell} : \mathsf{S}}
\qquad
\text{TR-VAR}\quad
\frac{\Delta;\Gamma \vdash \mathsf{x} : \mathsf{T}}{\Delta;\Gamma \vdash \mathsf{x} \rightsquigarrow \mathsf{x} : \mathsf{T}}
$$

Method transformation:

$$
\frac{\overline{\mathsf{X}}; \overline{\mathsf{x}} : \overline{\mathsf{T}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathsf{S}}{\mathsf{C}{<}\overline{\mathsf{X}}{>} \Vdash \mathsf{T}_0\ \mathsf{m}(\overline{\mathsf{T}}\ \overline{\mathsf{x}})^{\ell}\ \{\ \mathsf{return}\ \mathsf{e};\ \} \rightsquigarrow \mathsf{T}_0\ \mathsf{m}(\overline{\mathsf{T}}\ \overline{\mathsf{x}})^{\ell}\ \{\ \mathsf{return}\ \langle \mathsf{T}_0 \Leftarrow_{\ell} \mathsf{S}\rangle \mathsf{e}';\ \}}
$$

Class transformation:

$$
\frac{\mathsf{C}{<}\overline{\mathsf{X}}{>} \Vdash \overline{\mathsf{M}} \rightsquigarrow \overline{\mathsf{M}}'}{\mathsf{class}\ \mathsf{C}{<}\overline{\mathsf{X}}{>}\ \mathsf{extends}\ \mathsf{D}{<}\overline{\mathsf{S}}{>}\ \{\ \overline{\mathsf{T}}\ \overline{\mathsf{f}}; \mathsf{K}\ \overline{\mathsf{M}}\ \} \rightsquigarrow \mathsf{class}\ \mathsf{C}{<}\overline{\mathsf{X}}{>}\ \mathsf{extends}\ \mathsf{D}{<}\overline{\mathsf{S}}{>}\ \{\ \overline{\mathsf{T}}\ \overline{\mathsf{f}}; \mathsf{K}\ \overline{\mathsf{M}}'\ \}}
$$

Fig. 7. Coercion and guard insertion transformation.

requirements, we only retain enough information so that we can prove safety properties of the system, analogous to *threesomes* of Siek and Wadler [26][3].

Our approach is different from FGJ, which does not have to keep track of type argument values at runtime. We discuss our implementation strategy which improves efficiency by omitting some of the runtime type information in Section 4. In the formalisation we always keep them explicitly for clarity and simpler proofs.

Transforming method calls is more delicate due to mixed inheritance. It is not sufficient to insert coercions statically during transformation, since the runtime value may have been coerced from a different type (resulting in a type error) and since a subclass may override a method with a less-precise signature. Our strategy is to add argument co-

[3] As we only have one label for a value, we can blame *some* unsafe coercion performed on the value. It is easy to modify our semantics to keep track of, for example, all (or the last n) unsafe coercions performed on a value to get a more precise blame assignment, but with higher space requirements.

ercions based on static type of the target method and the argument expressions statically during transformation. Then we transform the method invocation into a *guarded method invocation* $\langle \mathtt{e.m}(\overline{\mathtt{e}})\rangle^{\ell}$ which performs additional coercions at runtime, based on runtime types of the receiver and the type arguments. We also include the label ℓ of the invocation (to assign blame).

3.7 Additional auxiliary functions and type relations

Method label lookup:

$$\frac{\mathtt{class\ C{<}\overline{X}{>}\ extends\ D{<}\overline{U}{>}}\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \} \quad \mathtt{T_0\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})^{\ell}\ \{\ \mathtt{return\ e;}\ \} \in \overline{\mathtt{M}}}{\mathit{mlabel}(\mathtt{m},\mathtt{C}) = \ell} \qquad \frac{\mathtt{class\ C{<}\overline{X}{>}\ extends\ D{<}\overline{U}{>}}\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \} \quad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mlabel}(\mathtt{m},\mathtt{C}) = \mathit{mlabel}(\mathtt{m},\mathtt{D})}$$

Translate type to superclass:

$$\mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{C}) = \mathtt{C{<}\overline{T}{>}} \qquad \frac{\mathtt{class\ C{<}\overline{X}{>}\ extends\ D{<}\overline{V}{>}}\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \}}{\mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{D}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D{<}\overline{V}{>}}}$$

$$\frac{\mathtt{class\ C{<}\overline{X}{>}\ extends\ D{<}\overline{S}{>}}\ \{\ldots\}}{\mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{E}) = \mathit{map}_{\uparrow}(\mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{D}),\mathtt{E})}$$

Translate type to subclass:

$$\mathit{map}_{\downarrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{C},\overline{\mathtt{S}}) = \mathtt{C{<}\overline{T}{>}} \qquad \frac{\mathtt{class\ C{<}\overline{X}{>}\ extends\ D{<}\overline{V}{>}}\ \{\ \overline{\mathtt{R}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \} \quad [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D{<}\overline{V}{>}} = \mathtt{D{<}\overline{S}{>}} \quad \mathtt{X}_i \notin \mathit{ftv}(\overline{\mathtt{V}}) \text{ implies } \mathtt{U}_i = \mathtt{T}_i}{\mathit{map}_{\downarrow}(\mathtt{D{<}\overline{S}{>}},\mathtt{C},\overline{\mathtt{T}}) = \mathtt{C{<}\overline{U}{>}}}$$

$$\frac{\mathtt{class\ C{<}\overline{X}{>}\ extends\ D{<}\overline{V}{>}}\ \{\ldots\} \quad \mathtt{D{<}\overline{W}{>}} = \mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{D}) \quad \mathit{map}_{\downarrow}(\mathtt{E{<}\overline{U}{>}},\mathtt{D},\overline{\mathtt{W}}) = \mathtt{D{<}\overline{S}{>}}}{\mathit{map}_{\downarrow}(\mathtt{E{<}\overline{U}{>}},\mathtt{C},\overline{\mathtt{T}}) = \mathit{map}_{\downarrow}(\mathtt{D{<}\overline{S}{>}},\mathtt{C},\overline{\mathtt{T}})}$$

Greatest common subtype (meet):

$$\mathit{meet}(\mathtt{T},\mathtt{T}) = \mathtt{T} \qquad \frac{\mathit{meet}(\mathtt{T},\mathtt{S}) = \mathtt{U}}{\mathit{meet}(\mathtt{S},\mathtt{T}) = \mathtt{U}} \qquad \mathit{meet}(\mathtt{T},\mathtt{dyn}) = \mathtt{T}$$

$$\frac{\mathtt{D{<}\overline{U}{>}} = \mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{D})}{\mathit{meet}(\mathtt{C{<}\overline{T}{>}},\mathtt{D{<}\overline{S}{>}}) = \mathit{map}_{\downarrow}(\mathtt{D{<}}\mathit{meet}(\overline{\mathtt{U}},\overline{\mathtt{S}})\mathtt{{>}},\mathtt{C},\overline{\mathtt{T}})}$$

Lowest common supertype (join):

$$\mathit{join}(\mathtt{T},\mathtt{T}) = \mathtt{T} \qquad \frac{\mathit{join}(\mathtt{T},\mathtt{S}) = \mathtt{U}}{\mathit{join}(\mathtt{S},\mathtt{T}) = \mathtt{U}} \qquad \mathit{join}(\mathtt{T},\mathtt{dyn}) = \mathtt{T} \qquad \frac{\mathtt{D{<}\overline{U}{>}} = \mathit{map}_{\uparrow}(\mathtt{C{<}\overline{T}{>}},\mathtt{D})}{\mathit{join}(\mathtt{C{<}\overline{T}{>}},\mathtt{D{<}\overline{S}{>}}) = \mathtt{D{<}}\mathit{join}(\overline{\mathtt{U}},\overline{\mathtt{S}})\mathtt{{>}}}$$

Fig. 8. Additional auxiliary functions (for target language).

In this section we discuss additional auxiliary functions and type relations used for defining the target language (Figures 8 and 9).

The function *mlabel* looks up the label of a method. It is used for assigning blame to a method. $\mathtt{map}_{\uparrow}$ and $\mathtt{map}_{\downarrow}$ translate the type arguments of a type $\mathtt{C{<}\overline{T}{>}}$ to a superclass or subclass of C, respectively. For example, assume we have a class definition like this:

$$\mathtt{T} \prec \mathtt{T} \qquad \mathtt{T} \prec \mathtt{dyn} \qquad \frac{\mathtt{D<}\overline{\mathtt{U}}\mathtt{>} = map_{\uparrow}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>},\mathtt{D}) \qquad \overline{\mathtt{U}} \prec_{=} \overline{\mathtt{S}}}{\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \prec \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}}$$

$$\mathtt{T} \prec_{co} \mathtt{T} \qquad \mathtt{T} \prec_{co} \mathtt{dyn} \qquad \frac{\mathtt{D<}\overline{\mathtt{U}}\mathtt{>} = map_{\uparrow}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>},\mathtt{D}) \qquad \overline{\mathtt{U}} \prec_{co} \overline{\mathtt{S}}}{\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \prec_{co} \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}}$$

Fig. 9. Type relations (narrowing) for the target language.

```
class C<X, Y> extends D<X, E<X>> { ... }
```

Now $\mathtt{map}_{\uparrow}(\mathtt{C<A,B<dyn>>,D}) = \mathtt{D<A,E<A>>}$. $\mathtt{map}_{\downarrow}$ requires an additional argument that provides values for type variables that cannot be mapped from the superclass, such as `Y` of `C` above (`Y` is not referred to in `D<X,E<X>>`). $\mathtt{map}_{\downarrow}(\mathtt{D<A,E<A>>,C,(dyn,B)}) = \mathtt{C<A,B>}$. Unlike $\mathtt{map}_{\uparrow}$, which is defined for all superclasses, not all types can be mapped to a subclass. For example, $\mathtt{map}_{\downarrow}(\mathtt{D<A,A>,C,}\overline{\mathtt{dyn}})$ is not defined.

We also define relation $\prec$ which, informally, determines whether a type is more precise or is a subtype. Using the above declaration of `C`, $\mathtt{C<A,dyn>} \prec \mathtt{D<A,E<A>>}$ and $\mathtt{C<A,B>} \prec \mathtt{D<dyn,dyn>}$. The related relation $\prec_{co}$ is similar to $\prec$, but it allows generic type arguments to vary covariantly. Thus if `A` is a subclass of `B`, $\mathtt{C<A,A>} \prec_{co} \mathtt{C<B,B>}$.

The following properties are useful when proving the soundness of our language. We omit several minor lemmas and trivial proofs; other proofs are sketchy. Detailed proofs for all lemmas and theorems in this paper (and additional minor lemmas) are available in the Appendix.

Definition 1. (Depth of a type) *$depth(\mathtt{T})$ is the depth of type* `T`. *$depth(\mathtt{dyn}) = depth(\mathtt{X}) = 1$. $depth(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = max(1 + max(depth(\mathtt{T}_i)), depth(\mathtt{map}_{\uparrow}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>},\mathtt{D})))$, if* `D` *is the direct superclass of* `C`. *We also assume that $max(depth(\bullet)) = 0$.*

Lemma 1. *$depth(\mathtt{map}_{\uparrow}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>},\mathtt{D})) \leq depth(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})$.*

Lemma 2. *$depth(\mathtt{map}_{\downarrow}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>},\mathtt{D},\overline{\mathtt{S}})) \leq max(depth(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}), depth(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}))$.*

Remark 1. The above properties are useful when proving properties of operations on types inductively. They also motivate our non-standard definition of type depth.

Lemma 3. *$\mathtt{map}_{\downarrow}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>},\mathtt{D},\overline{\mathtt{S}})$ is unique, if it exists.*

Lemma 4. (Transitivity of $\prec$, $\prec_{co}$ and $\prec_{=}$) *If $\mathtt{U} \prec \mathtt{T}$ and $\mathtt{T} \prec \mathtt{S}$, then $\mathtt{U} \prec \mathtt{S}$ (and similarly for $\prec_{co}$ and $\prec_{=}$).*

Proof. By induction on depth of `S`.

Lemma 5. *If $\mathtt{T} <: \mathtt{S}$, then $\mathtt{T} \prec \mathtt{S}$.*

Lemma 6. *If $\mathtt{T} \prec \mathtt{S}$, then $\mathtt{T} \prec_{co} \mathtt{S}$.*

The function *meet* evaluates to the greatest common subtype of two types, with respect to the $\prec_{co}$ relation. The *join* function evaluates the lowest common supertype of two types, but for a subtly different subtyping relation: `dyn` acts as the bottom type for *join*, whereas `dyn` is the top type for *meet* and $\prec_{co}$. The peculiarly-defined *join* serves a specific function in our semantics, which will become clear later in this section.

Definition 2. *The* inheritance hierarchy depth *between two classes* $\mathtt{C}$ *and* $\mathtt{D}$ *is written as* $\delta(\mathtt{C},\mathtt{D})$. $\delta(\mathtt{C},\mathtt{C}) = 0$ *and if* $\mathtt{C}$ *is a proper subclass of* $\mathtt{D}$ *and* $\mathtt{E}$ *is the direct superclass of* $\mathtt{C}$, $\delta(\mathtt{C},\mathtt{D}) = 1 + \delta(\mathtt{E},\mathtt{D})$.

Lemma 7. *If* $\mathtt{C{<}\overline{T}{>}} \prec_{\mathrm{co}} \mathtt{D{<}\overline{S}{>}}$ *and* $join(\mathtt{C{<}\overline{T}{>}},\mathtt{D{<}\overline{S}{>}}) = \mathtt{D{<}\overline{U}{>}}$, *then* $\mathtt{D{<}\overline{U}{>}} \prec_{=} \mathtt{D{<}\overline{S}{>}}$.

Proof. By induction on inheritance hierarchy depth between C and D.

Lemma 8. *If* $\mathtt{T} \prec \mathtt{S}$, *then* $\mathtt{T} <: join(\mathtt{T},\mathtt{S})$.

Lemma 9. *If* $\mathtt{T} \prec_{\mathrm{co}} \mathtt{S}$, *then* $meet(\mathtt{T},\mathtt{S}) = \mathtt{T}$.

Proof. By induction on maximum depth of $\mathtt{T}$ and $\mathtt{S}$.

Lemma 10. $meet(\mathtt{T},\mathtt{S}) \prec_{\mathrm{co}} \mathtt{T}$ *(if meet exists).*

Proof. By induction on maximum depth of $\mathtt{T}$ and $\mathtt{S}$.

Lemma 11. $meet(\mathtt{S}, meet(\mathtt{U},\mathtt{T})) = meet(\mathtt{S},\mathtt{T})$ *if* $\mathtt{S} \prec_{\mathrm{co}} \mathtt{U}$ *and the left hand side is defined.*

Proof. By induction on on maximum depth of $\mathtt{S}$, $\mathtt{T}$ and $\mathtt{U}$.

3.8 Target language typing

Expression typing:

TT-GINVK
$$\frac{\begin{array}{c} mtype(\mathtt{m},\mathtt{C{<}\overline{T}{>}}) = \overline{\mathtt{S}} \to \mathtt{S}_0 \\ \Delta;\Gamma \vdash \mathtt{e} : \mathtt{C{<}\overline{T}{>}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \end{array}}{\Delta;\Gamma \vdash \langle \mathtt{e.m}(\overline{\mathtt{e}})\rangle^{\ell} : \mathtt{S}_0}$$

TT-DYGINVK
$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathsf{dyn} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathsf{dyn}}}{\Delta;\Gamma \vdash \langle \mathtt{e.m}(\overline{\mathtt{e}})\rangle^{\ell} : \mathsf{dyn}}$$

TT-INVK
$$\frac{\begin{array}{c} mtype(\mathtt{m},\mathtt{C{<}\overline{T}{>}}) = \overline{\mathtt{S}} \to \mathtt{S}_0 \\ \Delta;\Gamma \vdash \mathtt{e} : \mathtt{C{<}\overline{T}{>}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \end{array}}{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}})^{\ell} : \mathtt{S}_0}$$

TT-DYINVK
$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathsf{dyn} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathsf{dyn}}}{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}})^{\ell} : \mathsf{dyn}}$$

TT-CREAT
$$\frac{\begin{array}{c} fields(\mathtt{C{<}\overline{S}{>}}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ \Delta \vdash \mathtt{U}\ \mathrm{OK} \qquad \Delta \vdash \mathtt{C{<}\overline{T}{>}}\ \mathrm{OK} \qquad \Delta \vdash \mathtt{C{<}\overline{S}{>}}\ \mathrm{OK} \\ \overline{\mathtt{T}} \prec_{\mathrm{co}} \overline{\mathtt{S}} \qquad \mathtt{C{<}\overline{T}{>}} \prec_{\mathrm{co}} \mathtt{U} \end{array}}{\Delta;\Gamma \vdash \langle \mathtt{U}\rangle \mathtt{new}\ \mathtt{C{<}\overline{T}} \Leftarrow \mathtt{\overline{S}{>}}_{\ell}(\overline{\mathtt{e}}) : \mathtt{U}}$$

TT-COERCE
$$\frac{\begin{array}{c} \Delta;\Gamma \vdash \mathtt{e} : \mathtt{S} \\ \Delta \vdash \mathtt{T}\ \mathrm{OK} \qquad \Delta \vdash \mathtt{S}\ \mathrm{OK} \end{array}}{\Delta;\Gamma \vdash \langle \mathtt{T} \Leftarrow_{\ell} \mathtt{S}\rangle \mathtt{e} : \mathtt{T}}$$

Method typing:

TT-METHOD
$$\frac{\begin{array}{c} \Delta = \overline{\mathtt{X}} \qquad \Delta \vdash \overline{\mathtt{T}}\ \mathrm{OK} \qquad \Delta \vdash \mathtt{T}_0\ \mathrm{OK} \\ \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C{<}\overline{X}{>}} \vdash \mathtt{e} : \mathtt{T}_0 \qquad validOverride(\mathtt{C{<}\overline{X}{>}}, \mathtt{m}, \overline{\mathtt{T}} \to \mathtt{T}_0) \end{array}}{\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})^{\ell}\ \{\ \mathtt{return}\ \mathtt{e};\ \}\ \mathrm{OK\ IN}\ \mathtt{C{<}\overline{X}{>}}}$$

Fig. 10. Typing for the target language. Rules that are equivalent to source language rules are omitted.

Figure 10 contains typing rules for the target language (this includes only those that are different from the source language). Note that the target language requires type equality instead of subtyping-or-consistency as in the source language. Rules for method overrides are equivalent to the source language, however. We add new rules for coercions and guarded method invocations (TT-GINVK, TT-DYGINVK and TT-COERCE).

3.9 Embedding untyped code

Typing embedded untyped code (source language):

$$\textsc{TE-Field}\quad \frac{\Delta;\Gamma \vdash \lfloor \mathsf{e} \rfloor : \mathsf{dyn}}{\Delta;\Gamma \vdash \lfloor \mathsf{e.f}^{\ell} \rfloor : \mathsf{dyn}}$$

$$\textsc{TE-Var}\quad \frac{\mathsf{x} \in \mathit{dom}(\Gamma)}{\Delta;\Gamma \vdash \lfloor \mathsf{x} \rfloor : \mathsf{dyn}}$$

$$\textsc{TE-Invk}\quad \frac{\Delta;\Gamma \vdash \lfloor \mathsf{e} \rfloor : \mathsf{dyn} \qquad \Delta;\Gamma \vdash \lfloor \overline{\mathsf{e}} \rfloor : \overline{\mathsf{dyn}}}{\Delta;\Gamma \vdash \lfloor \mathsf{e.m}(\overline{\mathsf{e}})^{\ell} \rfloor : \mathsf{dyn}}$$

$$\textsc{TE-Creat}\quad \frac{\#(\overline{\mathsf{e}}) = \mathit{nfields}(\mathsf{C}) \qquad \Delta;\Gamma \vdash \lfloor \overline{\mathsf{e}} \rfloor : \overline{\mathsf{dyn}}}{\Delta;\Gamma \vdash \lfloor \mathsf{new\ C}(\overline{\mathsf{e}})^{\ell} \rfloor : \mathsf{dyn}}$$

Transformation of embedded untyped code:

$$\textsc{TrE-Invk}\quad \frac{\Delta;\Gamma \vdash \lfloor \mathsf{e} \rfloor \rightsquigarrow \mathsf{e}' : \mathsf{dyn} \qquad \Delta;\Gamma \vdash \lfloor \overline{\mathsf{e}} \rfloor \rightsquigarrow \overline{\mathsf{e}}' : \overline{\mathsf{dyn}}}{\Delta;\Gamma \vdash \lfloor \mathsf{e.m}(\overline{\mathsf{e}})^{\ell} \rfloor \rightsquigarrow \langle \mathsf{e}'.\mathsf{m}(\overline{\mathsf{e}}')\rangle^{\ell} : \mathsf{dyn}}$$

$$\textsc{TrE-Creat}\quad \frac{\begin{array}{c}\mathit{cargs}(\mathsf{C}) = \overline{\mathsf{X}} \qquad \overline{\mathsf{U}} = \overline{\mathsf{dyn}} \qquad \#(\overline{\mathsf{U}}) = \#(\overline{\mathsf{X}}) \\ \mathit{fields}(\mathsf{C}\text{<}\overline{\mathsf{U}}\text{>}) = \overline{\mathsf{S}}\ \overline{\mathsf{f}} \qquad \Delta;\Gamma \vdash \lfloor \overline{\mathsf{e}} \rfloor \rightsquigarrow \overline{\mathsf{e}}' : \overline{\mathsf{T}}\end{array}}{\Delta;\Gamma \vdash \lfloor \mathsf{new\ C}(\overline{\mathsf{e}})^{\ell} \rfloor \rightsquigarrow \langle \mathsf{dyn} \Leftarrow_{\ell} \mathsf{C}\text{<}\overline{\mathsf{U}}\text{>}\rangle \langle \mathsf{C}\text{<}\overline{\mathsf{U}}\text{>}\rangle \mathsf{new\ C}\text{<}\overline{\mathsf{U}} \Leftarrow \overline{\mathsf{U}}\text{>}_{\circ}(\langle \overline{\mathsf{S}} \Leftarrow_{\ell} \overline{\mathsf{T}}\rangle \overline{\mathsf{e}}') : \mathsf{dyn}}$$

$$\textsc{TrE-Field}\quad \frac{\Delta;\Gamma \vdash \lfloor \mathsf{e} \rfloor \rightsquigarrow \mathsf{e}' : \mathsf{dyn}}{\Delta;\Gamma \vdash \lfloor \mathsf{e.f}^{\ell} \rfloor \rightsquigarrow \mathsf{e}'.\mathsf{f}^{\ell} : \mathsf{dyn}}$$

$$\textsc{TrE-Var}\quad \frac{\Delta;\Gamma \vdash \mathsf{x} : \mathsf{T}}{\Delta;\Gamma \vdash \lfloor \mathsf{x} \rfloor \rightsquigarrow \langle \mathsf{dyn} \Leftarrow \mathsf{T}\rangle \mathsf{x} : \mathsf{dyn}}$$

Fig. 11. Embedding untyped code.

Figure 11 contains additional typing and transformation rules for embedding untyped code within expressions using the $\lfloor \dots \rfloor$ form. Only minimal consistency checking is performed for embedded untyped code, and all generic instances are constructed with implicit dyn values for type arguments (TRE-CREAT). All values are coerced to dyn; for the evaluation rules discussed in this section, these coercions have no runtime effect.

An untyped class only defines untyped fields, method signatures and method bodies. When an untyped class extends a generic class, it should typically still support the same type arguments as the superclass to enable smooth interaction with typed code. For example, if the untyped `DList` class extends the typed `List<X>` class, we could declare `DList` like this:

```
class DList<X> extends List<X> { ... }
```

Now a `DList` instance created in untyped code has type `DList<dyn>` (by TRE-CREAT), and it can be coerced into `List<T>` for any `T`. A potential alternative definition

```
class DList extends List<dyn> { ... }
```

defines `DList` as a non-generic class, and it is only compatible with `List<dyn>`.

E-COERCE

$$\frac{meet(\mathsf{C<\overline{T}>},\mathsf{V}) = \mathsf{C<\overline{W}>} \qquad \text{if } \ell \neq \circ \text{ or } \mathsf{C<\overline{T}>} \prec \mathsf{V} \text{ then } \ell'' = \ell \text{ else } \ell'' = \ell'}{E[\langle \mathsf{V} \Leftarrow_{\ell'} \mathsf{U}\rangle\langle \mathsf{U}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}})] \longrightarrow E[\langle \mathsf{V}\rangle\mathsf{new\ C<\overline{W} \Leftarrow \overline{S}>}_{\ell''}(\overline{\mathsf{v}})]}$$

E-COERCE-FAIL

$$\frac{meet(\mathsf{C<\overline{T}>},\mathsf{V}) \neq \mathsf{C<\overline{W}>} \text{ or } meet(\mathsf{C<\overline{T}>},\mathsf{V}) \text{ is undefined}}{E[\langle \mathsf{V} \Leftarrow_{\ell} \mathsf{U}\rangle\langle \mathsf{U}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell'}(\overline{\mathsf{v}})] \longrightarrow \mathsf{blame}\ \ell}$$

E-GINVK

$$\frac{\begin{array}{c} join(map_{\uparrow}(\mathsf{C<\overline{U}>},\mathsf{D}),\mathsf{D<\overline{T}>}) = \mathsf{D<\overline{V}>} \qquad mtype(\mathsf{m},\mathsf{C<\overline{U}>}) = \overline{\mathsf{S}} \to \mathsf{S_0} \qquad mtype(\mathsf{m},\mathsf{D<\overline{V}>}) = \overline{\mathsf{R}} \to \mathsf{R_0} \\ mtype(\mathsf{m},\mathsf{D<\overline{T}>}) = \overline{\mathsf{W}} \to \mathsf{W_0} \qquad mtype(\mathsf{m},\mathsf{C<\overline{Q}>}) = \overline{\mathsf{P}} \to \mathsf{P_0} \qquad mtype(\mathsf{m},map_{\uparrow}(\mathsf{C<\overline{U}>},\mathsf{D})) = \overline{\mathsf{O}} \to \mathsf{O_0} \\ mlabel(\mathsf{m},\mathsf{C}) = \ell'' \qquad \mathsf{w} = \langle \mathsf{C<\overline{Q}>}\rangle\mathsf{new\ C<\overline{U} \Leftarrow \overline{Q}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{m}(\langle \overline{\mathsf{P}} \Leftarrow_{\ell} \overline{\mathsf{S}}\rangle\langle \overline{\mathsf{S}} \Leftarrow_{\circ} \overline{\mathsf{O}}\rangle\langle \overline{\mathsf{O}} \Leftarrow_{\ell} \overline{\mathsf{R}}\rangle\langle \overline{\mathsf{R}} \Leftarrow_{\ell'} \overline{\mathsf{W}}\rangle\overline{\mathsf{u}}) \end{array}}{E[\langle\langle \mathsf{D<\overline{T}>}\rangle\mathsf{new\ C<\overline{U} \Leftarrow \overline{Q}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{m}(\overline{\mathsf{u}})\rangle^{\ell'}] \longrightarrow E[\langle \mathsf{W_0} \Leftarrow_{\circ} \mathsf{R_0}\rangle\langle \mathsf{R_0} \Leftarrow_{\ell} \mathsf{O_0}\rangle\langle \mathsf{O_0} \Leftarrow_{\ell''} \mathsf{S_0}\rangle\langle \mathsf{S_0} \Leftarrow_{\ell} \mathsf{P_0}\rangle\mathsf{w}]}$$

E-DYGINVK

$$\frac{\begin{array}{c} mtype(\mathsf{m},\mathsf{C<\overline{U}>}) = \overline{\mathsf{S}} \to \mathsf{S_0} \\ mtype(\mathsf{m},\mathsf{C<\overline{V}>}) = \overline{\mathsf{W}} \to \mathsf{W_0} \qquad \mathsf{w} = \langle \mathsf{C<\overline{V}>}\rangle\mathsf{new\ C<\overline{U} \Leftarrow \overline{V}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{m}(\langle \overline{\mathsf{W}} \Leftarrow_{\ell} \overline{\mathsf{S}}\rangle\langle \overline{\mathsf{S}} \Leftarrow_{\ell'} \overline{\mathsf{dyn}}\rangle\overline{\mathsf{u}}) \end{array}}{E[\langle\langle \mathsf{dyn}\rangle\mathsf{new\ C<\overline{U} \Leftarrow \overline{V}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{m}(\overline{\mathsf{u}})\rangle^{\ell'}] \longrightarrow E[\langle \mathsf{dyn} \Leftarrow_{\circ} \mathsf{S_0}\rangle\langle \mathsf{S_0} \Leftarrow_{\ell} \mathsf{W_0}\rangle\mathsf{w}]}$$

E-DYGINVK-FAIL

$$\frac{mtype(\mathsf{m},\mathsf{C<\overline{V}>}) \text{ is undefined}}{E[\langle\langle \mathsf{dyn}\rangle\mathsf{new\ C<\overline{U} \Leftarrow \overline{V}>}_{\ell'}(\overline{\mathsf{v}}).\mathsf{m}(\overline{\mathsf{u}})\rangle^{\ell}] \longrightarrow \mathsf{blame}\ \ell}$$

E-INVK

$$\frac{mbody(\mathsf{m},\mathsf{C<\overline{X}>}) = \overline{\mathsf{x}}.\mathsf{e} \qquad cargs(\mathsf{C}) = \overline{\mathsf{X}}}{E[\langle \mathsf{C<\overline{S}>}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{m}(\overline{\mathsf{u}})] \longrightarrow E[[\overline{\mathsf{u}}/\overline{\mathsf{x}}, \langle \mathsf{C<\overline{S}>}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}})/\mathsf{this}, \overline{\mathsf{S}}/\overline{\mathsf{X}}]\mathsf{e}]}$$

E-FIELDACC

$$\frac{\begin{array}{c} join(\mathsf{C<\overline{T}>},\mathsf{D<\overline{U}>}) = \mathsf{D<\overline{Q}>} \\ fields(\mathsf{D<\overline{U}>}) = \overline{\mathsf{V}}\ \overline{\mathsf{f}} \qquad fields(\mathsf{D<\overline{Q}>}) = \overline{\mathsf{P}}\ \overline{\mathsf{f}} \qquad fields(\mathsf{C<\overline{T}>}) = \overline{\mathsf{W}}\ \overline{\mathsf{g}} \qquad fields(\mathsf{C<\overline{S}>}) = \overline{\mathsf{R}}\ \overline{\mathsf{g}} \end{array}}{E[\langle \mathsf{D<\overline{U}>}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{f}_i^{\ell'}] \longrightarrow E[\langle \mathsf{V}_i \Leftarrow_{\circ} \mathsf{P}_i\rangle\langle \mathsf{P}_i \Leftarrow_{\ell} \mathsf{W}_i\rangle\langle \mathsf{W}_i \Leftarrow_{\ell} \mathsf{R}_i\rangle\mathsf{v}_i]}$$

E-DYFIELDACC

$$\frac{fields(\mathsf{C<\overline{T}>}) = \overline{\mathsf{W}}\ \overline{\mathsf{g}} \qquad fields(\mathsf{C<\overline{S}>}) = \overline{\mathsf{R}}\ \overline{\mathsf{g}}}{E[\langle \mathsf{dyn}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}}).\mathsf{f}_i^{\ell'}] \longrightarrow E[\langle \mathsf{dyn} \Leftarrow_{\circ} \mathsf{W}_i\rangle\langle \mathsf{W}_i \Leftarrow_{\ell} \mathsf{R}_i\rangle\mathsf{v}_i]}$$

E-DYFIELDACC-FAIL

$$\frac{fields(\mathsf{C<\overline{S}>}) = \overline{\mathsf{U}}\ \overline{\mathsf{g}} \qquad \mathsf{f} \notin \overline{\mathsf{g}}}{E[\langle \mathsf{dyn}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell'}(\overline{\mathsf{v}}).\mathsf{f}^{\ell}] \longrightarrow \mathsf{blame}\ \ell}$$

E-CAST

$$\frac{meet(\mathsf{C<\overline{T}>},\mathsf{D<\overline{V}>}) = \mathsf{C<\overline{W}>} \qquad \text{if } \ell \neq \circ \text{ or } \mathsf{C<\overline{T}>} \prec \mathsf{D<\overline{V}>} \text{ then } \ell'' = \ell \text{ else } \ell'' = \ell'}{E[(\mathsf{D<\overline{V}>})^{\ell'}\langle \mathsf{U}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}})] \longrightarrow E[\langle \mathsf{D<\overline{V}>}\rangle\mathsf{new\ C<\overline{W} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}})]}$$

E-DYNCAST

$$E[(\mathsf{dyn})\langle \mathsf{U}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}})] \longrightarrow E[\langle \mathsf{dyn}\rangle\mathsf{new\ C<\overline{T} \Leftarrow \overline{S}>}_{\ell}(\overline{\mathsf{v}})]$$

Fig. 12. Evaluation.

3.10 Evaluation

Evaluation rules for the language are shown in Figure 12. The most interesting rules are for method invocation (evaluated in two steps, E-GINVK and E-INVK) and coercions (E-COERCE-*).

Coercions A coercion $\langle \mathtt{S} \Leftarrow_\ell \mathtt{T}\rangle\mathtt{e}$ coerces the value of expression e from type T to S. The validity of coercions between generic types cannot generally be determined immediately; the label ℓ is blamed if a later, lazy type check fails due to this coercion.

Definition 3. (Safe coercions and casts) *A coercion or cast is safe if it cannot be a target of blame (i.e. it preserves the blame label of the target value) and if it always succeeds.*

If $\mathtt{T} \prec \mathtt{S}$, the coercion is *safe* and has no runtime effect, other than changing the current type of the target value to T. A coercion is safe if, for example, the coercion source type T is a subtype of the target type S, or if the target type is dyn.

When coercing generic values, the type arguments may be changed in the coercion. For example, the coercion $\langle \mathtt{D<C>} \Leftarrow_\ell \mathtt{D<dyn>}\rangle\langle \mathtt{D<dyn>}\rangle\mathtt{new\ D<dyn} \Leftarrow \mathtt{dyn>}_\circ(\overline{\mathtt{v}})$ evaluates to the value $\langle \mathtt{D<C>}\rangle\mathtt{new\ D<C} \Leftarrow \mathtt{dyn>}_\ell(\overline{\mathtt{v}})$, with type argument C instead of dyn and a new label ℓ, but otherwise identical to original value. We change the instance label to keep track of the location (label) of the coercion. This allows us to assign blame to the coercion if we later detect a type error lazily.

For non-generic types, the coercion behaves like a cast that evaluates to blame ℓ if unsuccessful (a special case of E-COERCE and E-COERCE-FAIL when C has no type arguments).

Valid blame targets

Definition 4. (Valid blame target) *The label ℓ is valid target for blame if it refers to one of the following constructs:*

1. *an invocation which uses unsafe coercion for arguments*
2. *a* new *expression which uses unsafe coercion for arguments*
3. *an invocation or field access where the receiver type is not precise, e.g.* dyn *or* C<dyn>
4. *a method body which uses an unsafe coercion for the return value*
5. *a method override that is unsafe (return value type of the signature is less precise than in the overridden method)*
6. *a downcast.*

In particular, the empty label is not a valid blame target. We later prove that if evaluation terminates in blame ℓ, the blame label is always valid[4].

Method invocation To fully evaluate a method call, we perform *five* sets of coercions for the arguments. We explain each of these coercions and their blame labels separately.

Transformation produces the first set of coercions (TR-INVK):

$\langle \overline{\mathtt{S}} \Leftarrow_\ell \overline{\mathtt{T}}\rangle$ Coerce from static argument expression types to static formal argument types, based on the static receiver type. Blame the call expression (ℓ) if the coercion fails: in this case, the method was called with an argument with type that was not compatible (at runtime) with the formal arguments, and the coercion for the argument was not safe. For example, a method with signature C m(C x) may be called with argument expression with type dyn. The coercion $\langle \mathtt{C} \Leftarrow \mathtt{dyn}\rangle$ may fail at runtime, if the argument has type that is incompatible with C.

[4] A complete, practical language would have additional valid blame targets, such as assignment statements from a less precise to a more precise type

The additional four are produced during evaluation (E-GINVK). The first three coerce from static formal argument types to runtime method argument types based on the greatest lower bound of type arguments. The final coercion coerces to the argument types based on the original type argument values, used during object creation.

$\langle \overline{\mathsf{R}} \Leftarrow_{\ell'} \overline{\mathsf{W}} \rangle$ Coerce from static formal argument type to method argument types generated using the greatest lower bound of type arguments. We do this in three coercions, each blaming a different target. This the first step; it blames the invocation, and if $\overline{\mathsf{R}} \neq \overline{\mathsf{W}}$, the static receiver type is less precise than the runtime type (e.g. `A<dyn>` vs. `A<B>`), by properties of *join*.

$\langle \overline{\mathsf{O}} \Leftarrow_{\ell} \overline{\mathsf{R}} \rangle$ The third coercion (the second step in the coercion mentioned above) coerces to the signature based on the greatest lower bound of type arguments and the static receiver class. This is significant, for example, if a value with initial type `A<dyn>` is coerced into `A<B>` and later again to `A<dyn>`. Now both the original and current types of the receiver are the same (`A<dyn>`) and thus would not cause blame. We also use the greatest lower bound `A<B>` to make sure that the arguments are compatible with *all coercions* that have affected the value. This coercion blames the coercion that produced the instance. If the coercion fails, there must have been an unsafe generic coercion.

$\langle \overline{\mathsf{S}} \Leftarrow_{\circ} \overline{\mathsf{O}} \rangle$ This coercion coerces from the method signature based on the *static* class and *runtime* type argument values ($\overline{\mathsf{O}}$) to the method signature based on the *runtime* class and *runtime* type argument values ($\overline{\mathsf{S}}$). Since the type arguments for both signatures are identical, the differences in $\overline{\mathsf{S}}$ and $\overline{\mathsf{O}}$ can only derive from a less specific type signature – by definition of *validOverride*, the signature of the overriding method must be less precise than the original one. Therefore, the coercion is safe and it is correct to use the empty blame label.

$\langle \overline{\mathsf{P}} \Leftarrow_{\ell} \overline{\mathsf{S}} \rangle$ The final coercion coerces to the original method signature of the value, before any coercions. This coercion blames the coercion that produced the instance. If this coercion fails, there was an invalid unsafe generic coercion.

We also perform four coercions for the method return value, in order to assign blame correctly and in order to catch all type errors. These mirror the coercions for arguments. However, if a method override has a less specific return value than the overridden method, we may have to blame the override as unsafe (using label ℓ'').

The E-GINVK rule transforms the guarded method call into an ordinary invocation. The evaluation of the invocation finishes with E-INVK, which is standard (unless some coercions failed).

Let's assume we create an untyped list that contains both `String` and `Foo` instances and pass it to a method expecting a `List<String>` argument (statically). We perform the coercion $\langle$`List<String>` $\Leftarrow_{\ell}$ `List<dyn>`$\rangle$ without looking at the list contents (we wish a coercion to have effectively constant execution time). Now when the method body calls the `head` method of the list object (which returns a value with the type variable type, in this case `String`), we do not know statically whether the return value matches the runtime type variable (`String`). So, at runtime, we generate a coercion for the return value from `dyn` to `String`.

Now since we assumed that the list contains also `Foo` objects, we may get a failed coercion if the method accesses one of those items. In this case we use the blame label to blame the location that caused the original coercion, which coerced `List<dyn>` to `List<String>`; an unsafe coercion was the source of the error.

Field access To evaluate field access we coerce field types with type variable components $\mathtt{X}$ using the actual values of type variables (E-FIELDACC), and blame the coercion if this fails. This is analogous to the coercion of method return values above.

Untyped receiver We have separate rules for invocations and field access with a dyn receiver. They are similar to rules to non-dyn receivers, only simpler.

Casts Casts are evaluated similarly to coercions. We can also blame failed downcasts.

3.11 Properties

We show how to prove the soundness of the target language. We first prove the soundness of transformation, followed by ordinary progress and type preservation lemmas. Our main result is that if evaluation terminates with blame, we always point to a valid blame target, as defined in Definition 4.

Lemma 12. (Type preservation during transformation) *If* $\Delta,\Gamma \vdash \mathtt{e} : \mathtt{T}$ *(using source language typing rules) and* $\Delta,\Gamma \vdash \mathtt{e} \rightsquigarrow \mathtt{e}' : \mathtt{T}$*, then* $\Delta,\Gamma \vdash \mathtt{e}' : \mathtt{T}$ *(using alternative typing rules).*

Proof. By induction on transformation relation.

Lemma 13. *If* $\Delta,\Gamma \vdash \mathtt{e} : \mathtt{T}$ *(using source language typing rules) and* $\Delta,\Gamma \vdash \mathtt{e} \rightsquigarrow \mathtt{e}' : \mathtt{T}$*, then the label of the coercion is a valid blame target (by Definition 4) for any unsafe coercion in* $\mathtt{e}'$.

Proof. By induction on transformation relation.

The above two lemmas justify that our transformation rules are sound with respect to typing rules. In the rest of this section, we implicitly assume that a well-typed source program has been transformed to the target language. This provides a suitable base for inductive proofs. If we did not make this assumption, the initial state might include new expressions or coercions with invalid blame labels.

Lemma 14. (Progress) *If* $\Delta,\Gamma \vdash \mathtt{e} : \mathtt{T}$ *(using target language typing rules) and* $\mathtt{e} \longrightarrow^{*} \mathtt{e}'$ *and* $\mathtt{e}' \neq \mathtt{blame}\ \ell$*, then either* $\mathtt{e}'$ *is a value, or* $\mathtt{e}' \longrightarrow \mathtt{e}''$ *(unless there is a failed downcast).*

Proof. By induction on evaluation relation.

Lemma 15. (Type preservation) *If* $\Delta,\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^{*} \mathtt{e}'$*, then* $\Delta,\Gamma \vdash \mathtt{e}' : \mathtt{T}$.

Proof. By induction on evaluation relation. Also use Lemma 16 (below) for case E-INVK.

Lemma 16. (Substitution lemma) *If* $\mathit{mbody}(\mathtt{m},\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}) = \overline{\mathtt{x}}\ .\ \mathtt{e}$*,* $\mathit{cargs}(\mathtt{C}) = \overline{\mathtt{X}}$ *and*

$$\langle \mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}\rangle \mathtt{new\ C}\texttt{<}\overline{\mathtt{T}} \Leftarrow \overline{\mathtt{S}}\texttt{>}_{\ell}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}})$$

is well-typed, then also

$$[\overline{\mathtt{u}}/\overline{\mathtt{x}}, \langle \mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}\rangle \mathtt{new\ C}\texttt{<}\overline{\mathtt{T}} \Leftarrow \overline{\mathtt{S}}\texttt{>}_{\ell}(\overline{\mathtt{v}})/\mathtt{this}, \overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{e}$$

is well-typed.

Proof. By induction on depth of e.

Note that the progress lemma does not state anything about the validity of blame labels. In particular, it does not assure us that programs never blame the empty blame label $\circ$. Traditional progress and preservation lemmas are thus not sufficient for proving the soundness of our language. We additionally show that if evaluation terminates with blame, we assign blame to a valid target.

Theorem 1. (Valid blame assignment) *If* $\Delta, \Gamma \vdash \mathsf{e} : \mathsf{T}$ *and* $\mathsf{e} \longrightarrow^* \mathtt{blame}\ \ell$*, then* ℓ *is a valid blame target, according to Definition 4.*

Proof. By induction on evaluation relation. We ensure that evaluation preserves several additional properties; we can prove their preservation separately in the induction step:

1. All unsafe coercions in e have a valid blame label (as in Lemma 4).
2. For each new expression with an empty label (of form $\mathtt{<U>new\ C<}\overline{\mathsf{T}} \Leftarrow \overline{\mathsf{S}}\mathtt{>}_\circ(\ldots)$), $\mathtt{C<}\overline{\mathsf{T}}\mathtt{>} \prec \mathsf{U}$.
3. If the blame label of a new expression in e is non-empty, it has been through an unsafe coercion or an unsafe downcast, and the blame label refers to an unsafe coercion or a downcast.
4. If the label $\ell = \circ$ for $\mathtt{<U>new\ C<}\overline{\mathsf{T}} \Leftarrow \overline{\mathsf{S}}\mathtt{>}_\ell(\ldots)$ in e, then $\overline{\mathsf{S}} = \overline{\mathsf{T}}$.

Case E-GINVK is the most complex. We can show separately for each coercion that either the coercion gets a label that is valid blame target, or the coercion is safe (by Lemma 18) and can never fail. We also use above properties 2–4 and the properties of *join* and other auxiliaries. Cases E-DYGINVK and E-FIELDACC are similar to E-GINVK. For E-COERCE we use Lemma 17. Casts are similar to coercions.

Lemma 17. (Safe coercion) *Coercion* $\langle \mathsf{T} \Leftarrow_\ell \mathsf{S}\rangle\langle \mathsf{S}\rangle\mathtt{new\ C<}\overline{\mathsf{U}} \Leftarrow \overline{\mathsf{V}}\mathtt{>}_{\ell'}(\ldots)$ *is safe if* $\mathtt{C<}\overline{\mathsf{U}}\mathtt{>} \prec \mathsf{T}$ *or if* $\mathsf{S} \prec \mathsf{T}$.

Proof. Use properties of *meet*.

Lemma 18. (Safe coercion) *Coercion* $\langle \mathsf{T} \Leftarrow_\ell \mathsf{S}\rangle$ *is safe if* $\mathsf{S} \prec \mathsf{T}$.

Proof. This follows from the properties used in the valid blame theorem.

Definition 5. (Fully-typed programs) *A program is fully-typed if it does not contain the type* dyn.

Theorem 2. *All coercions are safe when evaluating a fully-typed program, and the evaluation always terminates without blame if there are no unsafe downcasts.*

Proof. This is essentially a special case of the valid blame assignment theorem proof.

We also introduce two simpler forms of the semantics that are equivalent (bisimilar) to the original semantics. They form a better basis for an efficient implementation, but the original semantics allows a simpler type safety proof. The definition of the semantics and the bisimilarity is very straightforward, but would require a lengthy detailed presentation, so we omit a formal definition.

Definition 6. (Alternative semantics) *We informally define an alternative semantics for the target language with these differences:*

1. *Do not keep track of the current type of each value. Values thus have form* $\texttt{new C<}\overline{\texttt{T}} \Leftarrow \overline{\texttt{S}}\texttt{>}_\ell(\dots)$ *instead of* $\langle\texttt{U}\rangle\texttt{new C<}\overline{\texttt{T}} \Leftarrow \overline{\texttt{S}}\texttt{>}_\ell(\dots)$.
2. *Attach the receiver type to guarded method invocations, since the evaluation rules* E-GINVK *and* E-DYGINVK *depend on it. A guarded method invocation is thus of form* $\langle\texttt{e.m}(\overline{\texttt{e}})\rangle^\ell_\texttt{U}$*, if the receiver would be of form* $\langle\texttt{U}\rangle\texttt{new}\ \dots$ *in the original semantics.*
3. *Do not keep track of the coercion source types. Coercions are of form* $\langle\texttt{T}_\ell\rangle$ *instead of* $\langle\texttt{T} \Leftarrow_\ell \texttt{S}\rangle$.

Theorem 3. *The alternative semantics is bisimilar to the original semantics.*

Proof. Inductive bisimilarity proof of the evaluation relations.

The coercion source type is still sometimes useful to retain, as we can use it to omit trivial coercions. An implementation can use the source type whenever it allows more efficient evaluation and erase it elsewhere.

Definition 7. (Semantics without blame labels) *We informally define an alternative semantics for the target language with these differences from the first alternative semantics:*

1. *Do not keep track of the blame label of values. Values are of form* $\texttt{new C<}\overline{\texttt{T}} \Leftarrow \overline{\texttt{S}}\texttt{>}(\dots)$.
2. *Update evaluation accordingly to not process blame labels, and each rule result* $\texttt{blame}\ \ell$ *is replaced with* `blame`*, without a blame label.*

Theorem 4. *The second alternative semantics is bisimilar to the original semantics, modulo reported blame labels.*

Proof. Inductive bisimilarity proof of the evaluation relations.

The above theorem allows an implementation omit blame tracking in release builds, for example. This may result in higher runtime efficiency.

4 Implementation

A naive implementation of our semantics uses reflection to invoke methods, to access fields and to select which coercions to perform. Clearly a more efficienct implementation would be preferable. In this section we explain how to implement the language efficiently.

Our technique is general enough to support several different compilation strategies:

1. compilation to a custom virtual machine optimised for the language
2. compilation to an existing typed virtual machine, such as the JVM
3. compilation to an untyped language, such as JavaScript.

The implementation performs a program transformation (this transformation is in addition to the transformation described in Section 3) that generates wrapper methods and classes for performing necessary coercions. Many coercions can be trivially optimised away during transformation. We describe the transformation in the following subsections.

In addition to this transformation, we also need to provide an implementation of the coercion operation. Coercions between non-generic types `C` and `D` are simply runtime type checks. For more complex coercions the implementation can use caching to avoid repeatedly evaluating potentially expensive *meet* and *join* operations. We can easily optimise away coercions that have no runtime effect. For example, the coercion $\langle \mathtt{T} \Leftarrow_{\ell} \mathtt{S} \rangle$ can be omitted if $\mathtt{S} <: \mathtt{T}$. All coercions in fully-typed programs are of this kind.

We focus on three aspects of the implementation in the following subsections: genericity, mixed inheritance and dyn receiver types. Our implementation technique introduces very little runtime overhead for pure untyped and typed code, when compared to an implementation of a pure typed on untyped language, respectively[5].

4.1 Implementing genericity

Our semantics support multiple references to the same object with different values of type variables. We can implement this by having one shared instance with the original type variables (which are fixed) and any number of wrapper objects, each of which stores the greatest lower bound of type variables due to coercions, a label and a reference to the original object.

So, for an object of type $\mathtt{new\ C{<}\overline{S} \Leftarrow \overline{T}{>}(\dots)}$ we would have an object with type $\mathtt{C{<}\overline{T}{>}}$ and a separate wrapper object representing type $\mathtt{C{<}\overline{S}{>}}$. The wrapper object coerces values with type variable type first to the greatest lower bound types ($\overline{\mathtt{S}}$) and then to the original type variable types ($\overline{\mathtt{T}}$); the unwrapped object does not need to perform any coercions for function arguments or return values. The wrapper class implements the same public interface as the original class.

However, in pure typed and untyped sections of code there are no coercions that change type variables; these only happen in mixed scenarios. We can implement instances like these, of form $\langle \mathtt{D{<}\overline{S}{>}} \rangle \mathtt{new\ C{<}\overline{T} \Leftarrow \overline{T}{>}_{\circ}(\dots)}$ such that $\mathtt{C{<}\overline{T}{>}} <: \mathtt{D{<}\overline{S}{>}}$ in the original semantics, without any wrappers or coercions. Thus only programs that actively use the features of gradual typing experience performance overhead from wrappers. We expect that a significant fraction of values would never escape the *island* of typed or untyped code where they are created and thus would require no wrappers.

4.2 Implementing mixed inheritance

Method overrides may have less precise signatures than original superclass methods. Therefore even when calling a method with a precise signature, we may have to coerce some of the arguments or the return value at runtime, in case a subclass has overridden the method with a less precise type. As mentioned above, even coercions to a less precise types may now generate wrapper objects.

This can be implemented efficiently by generating multiple variants of a method during compilation, one for the original method and one for each different type signature of an overriding method. When calling a method, the variant is picked based on the static receiver type. The least specific variant contains the method implementation and the other variants simply coerce the arguments and call the least specific variant.

For example, assume class C defines method $\mathtt{E\ m(A{<}B{>}\ x)}$. The compiler changes the method signature into $\mathtt{D\ m^{C}(A{<}B{>}\ x)}$, but keeps the method body intact (the method body

[5] We generate some additional method variants and wrapper classes that are only used for interaction between untyped and typed code. Public fields require access methods instead of direct access. Additionally, we need to keep track of blame labels if the implementation supports blame.

is never changed in this transformation). Now class D extends C and overrides m with signature dyn m(dyn x). The compiler changes this method signature to dyn $\mathtt{m}^{\mathtt{D}}$(dyn x) and also generates another method that actually overrides the superclass method $\mathtt{m}^{\mathtt{C}}$ and calls $\mathtt{m}^{\mathtt{D}}$:

$$\mathtt{E\ m^{C}(E\ x)\ \{\ return\ } \langle \mathtt{E} \Leftarrow_{\ell} \mathtt{dyn}\rangle \mathtt{this.m^{D}}(\langle \mathtt{dyn} \Leftarrow_{\circ} \mathtt{A{<}B{>}}\rangle \mathtt{x});\ \}$$

If the receiver type in a call to method m is C, the method name is changed to $\mathtt{m}^{\mathtt{C}}$, and similarly for receiver D. If E inherits D, it has to override the method with signature identical to $\mathtt{m}^{\mathtt{D}}$; thus the method name in E would still be $\mathtt{m}^{\mathtt{D}}$.

4.3 Implementing dyn receiver types

If the receiver type is dyn, the above techniques do not suffice. Similar to mixed inheritance, we can generate a new method variant that we can call when the receiver type is dyn. It simply coerces the arguments and the return value from and to dyn, respectively. For the method m of C in the above example we would generate method $\mathtt{m}^{\mathtt{dyn}}$:

$$\mathtt{dyn\ m^{dyn}(dyn\ x, Label\ } \ell)\ \mathtt{\{\ return\ } \langle \mathtt{dyn} \Leftarrow_{\ell} \mathtt{E}\rangle \mathtt{this.m^{C}}(\langle \mathtt{A{<}B{>}} \Leftarrow_{\ell} \mathtt{dyn}\rangle \mathtt{x});\ \}$$

The caller passes the label of the invocation as the ℓ argument. If any of the coercions fails, we blame the call, similar to our original semantics (E-DYGINVK). If we call a method of a generic class, the wrapper method may need the value of type variables to perform coercions. It can look them up from the generic instance or the wrapper object.

When invoking a method with a dyn receiver, our implementation only needs to look up the method; it does not have to look up the signature and generate coercions dynamically. If the called method is untyped, all of the coercions are trivial ($\langle \mathtt{dyn} \Leftarrow_{\ell} \mathtt{dyn}\rangle$) and can be optimised away, and the untyped method wrapper can thus be an alias of the original method.

4.4 Additional implementation issues

Adding support for assignment, control structures and different member visibility levels to our language is straightforward. An implementation can support primitive types encoded using a uniform object encoding or with automatic boxing and unboxing operations.

Some basic features cannot be implemented using the most direct implementation techniques. As wrappers must support evaluating public field accesses, these field references cannot be implemented using direct field access, as in JVM for example, but we have to translate them to method calls (Ina and Igarashi [19] also mention a similar issue). Additionally, object equality cannot be simply a pointer comparison, as different wrapped instances of an object should be equal to each other and the unwrapped object.

An implementation that compiles to Java bytecode or JavaScript may thus not be able to seamlessly interact with native Java or JavaScript code. The implementation may need a separate interoperability layer, unless the virtual machine is modified to natively support gradual typing using our implementation technique.

5 Related work

Gradual typing without blame Siek and Taha introduced the concept *gradual typing* and have presented a functional language [25] and an object-oriented language (based on structural subtyping) with gradual typing [24]. Neither of these support genericity or blame, however. They do not present an efficient implementation technique. Herman et al. [16] and Siek and Wadler [26] discuss techniques for reducing the memory usage of gradual typing. Ina and Igarashi [19] present a form of gradual typing for a Java-like language with generics. Unlike our work, they do not support mixed inheritance or blame, and their language does not allow many coercions supported by our language, such as from runtime type `C<dyn>` to `C<D>`. They can give a stronger type safety guarantee for typed code, but at the cost of reduced flexibility.

Blame Findler and Felleisen use blame to report the source of runtime errors when using contracts with higher-order functions [9]. *Semantic casts* [10] allow coercions between different nominal types with structural similarity but with different contracts. The approach can properly assign blame when a contract is violated. Tobin-Hochstadt and Felleisen [27] track blame in a functional language that allows interaction between typed and untyped modules. Wadler and Findler's blame calculus [30] supports fine-grained mixing of untyped and typed code, still in a functional setting. Ahmed et al. [2] extend the functional blame calculus with parametric polymorphism.

A technique for Java-Scheme interoperability by Gray et al. [15] supports blame, but it does not have genericity or mixed inheritance. A later system [13] supports blame with mixed inheritance between an untyped and a typed language. Unlike our language, this approach does not support genericity or fine-grained mixing of typed and untyped code, and Gray does not present an efficient implementation technique. Gray also presents an approach that supports mixed inheritance between a class-based and a prototype-based language [14], with similar differences to our work.

Unsound approaches Several authors have proposed approaches for combining static and dynamic typing in a language that are not sound but retain some benefits of static typing. Optional and pluggable [5] type systems do not affect runtime semantics. Strongtalk [6] is a variant of Smalltalk with an optional type system. Pure optional type systems do not track type errors arising from interaction between typed and untyped code, unlike a gradual type system.

Lehtosalo and Greaves [21] explain how to detect and log runtime type errors as soft errors in an optionally-typed language without otherwise affecting evaluation semantics. Their language does not support mixed inheritance or genericity. Dart [8] is a recent language that combines optional typing with an alternative *checked mode* that performs runtime type checking resembling gradual typing. At least the current work-in-progress version of Dart has an unsound type system and semantics even for fully typed programs in the checked mode, and it does not support blame. Dart supports reified generics.

Thorn [4] supports "like" types [31] that are checked statically but do not have runtime type safety, in addition to concrete types that are type-safe. Thorn does not seem to have generics or mixed inheritance.

Static typing for untyped languages PRuby [12] uses profile-guided static analysis with optional type annotations to find errors in Ruby programs that use dynamic language features. The analysis is not modular or sound, and PRuby does not support fine-grained evolution from untyped to typed code.

Typed Scheme[6] [28] a variant of Scheme with a static type system. It supports parametric polymorphism and interaction between typed and untyped modules, but parametrically-polymorphic functions cannot be exported to untyped code. The authors argue that the type system is sound.

Restricted mixed typing Several languages support using untyped values in a typed language but do not allow seamless integration of typed and untyped code. Abadi et al. [1] formalise a typed language with type `Dynamic` and an explicit `typecase` construct. C# has the type `dynamic` [3] and genericity, but it does not support mixed inheritance or coercions between different generic types such as `C<dynamic>` and `C<D>`.

Genericity Many object-oriented languages support genericity. GJ [7] and Java generics are based on type erasure: values of type parameters are not available at runtime. Java-style generics are not type-safe when using reflection or unsafe casts. FGJ [17] is a formalisation of a subset of GJ; our formalisation was inspired by FGJ. The .NET Framework supports reified generics [20]. Unlike our language, .NET does not support coercions between generic types of different typing precision.

Java and other languages have type system support [18, 29] for covariant and contravariant compatibility between generic types, which we refer to as *variance*. This requires explicit variance declarations for type parameters. Our language supports variance in coercions (and casts) for all generic types. This frees the programmer from the need of providing variance annotations, at the expense of potential runtime errors as result of unsafe coercions or casts. Fully-typed code must use explicit casts for potentially unsafe coercions.

Hybrid type checking Hybrid type checking [11] uses static analysis to check detailed function contracts whenever possible and falls back to dynamic checking when this is not possible.

6 Conclusions and future work

We have shown how to combine gradual typing, mixed nominal inheritance, genericity and blame in a single language. We formalised a core language, proved that the language is sound and presented a technique for implementing the language efficiently. This work enables the design and implementation of practical, object-oriented languages with support for gradual typing.

We are currently working on an implementation of the techniques for a new programming language that aims at very close resemblance to Python. Additional future work includes the integration of various new language features such as higher-order and generic functions [2], bounded quantification and retroactive supertype definition.

[6] Typed Scheme is now called Typed Racket.

References

1. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Transactions on Programming Languages and Systems 13(2), 237–268 (April 1991)
2. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 201–214 (2011)
3. Bierman, G., Meijer, E., Torgersen, M.: Adding Dynamic Types to C#. In: ECOOP 2010 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 6183, chap. 5, pp. 76–100 (2010)
4. Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strniša, R., Vitek, J., Wrigstad, T.: Thorn: robust, concurrent, extensible scripting on the JVM. In: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object-oriented programming systems languages and applications. pp. 117–136 (2009)
5. Bracha, G.: Pluggable type systems. In: OOPSLA Workshop on Revival of Dynamic Languages (2004)
6. Bracha, G., Griswold, D.: Strongtalk: typechecking Smalltalk in a production environment. In: OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. pp. 215–230 (1993)
7. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: adding genericity to the Java programming language. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 183–200. OOPSLA '98 (1998)
8. Bracha, G., et al.: The Dart programming language. `http://www.dartlang.org/` (2011)
9. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. vol. 37, pp. 48–59 (September 2002)
10. Findler, R.B., Flatt, M., Felleisen, M.: Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In: ECOOP 2004 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3086, chap. 17, pp. 614–639 (2004)
11. Flanagan, C.: Hybrid type checking. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 245–256 (2006)
12. Furr, M., An, J.h.D., Foster, J.S.: Profile-guided static typing for dynamic scripting languages. In: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object-oriented programming systems languages and applications. pp. 283–300 (2009)
13. Gray, K.: Safe Cross-Language Inheritance. In: ECOOP 2008 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 5142, chap. 4, pp. 52–75 (2008)
14. Gray, K.E.: Interoperability in a Scripted World: Putting Inheritance & Prototypes Together. In: International Workshop on Foundations of Object-Oriented Languages (FOOL '10) (2010)
15. Gray, K.E., Findler, R.B., Flatt, M.: Fine-grained interoperability through mirrors and contracts. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 231–245 (2005)
16. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. Trends in Functional Programming (2007)
17. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
18. Igarashi, A., Viroli, M.: On Variance-Based Subtyping for Parametric Types. In: ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming. pp. 441–469 (2002)
19. Ina, L., Igarashi, A.: Gradual typing for generics. In: Proceedings of the 2011 ACM international conference on Object-oriented programming systems languages and applications. pp. 609–624 (2011)

20. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation. pp. 1–12. PLDI '01 (2001)
21. Lehtosalo, J., Greaves, D.J.: Language with a pluggable type system and optional runtime monitoring of type errors. In: STOP 2011: International workshop on Scripts to Programs (2011)
22. Ousterhout, J.K.: Scripting: Higher-Level Programming for the 21st Century. Computer 31(3), 23–30 (Mar 1998)
23. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
24. Siek, J., Taha, W.: Gradual typing for objects. In: ECOOP '07. pp. 2–27 (2007)
25. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (2006)
26. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: STOP '09: Proceedings for the 1st workshop on Script to Program Evolution. pp. 34–46 (2009)
27. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. pp. 964–974 (2006)
28. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 395–406 (2008)
29. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: SAC '04: Proceedings of the 2004 ACM symposium on Applied computing. pp. 1289–1296 (2004)
30. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems. pp. 1–16 (2009)
31. Wrigstad, T., Nardelli, F.Z., Lebresne, S., Östlund, J., Vitek, J.: Integrating typed and untyped code in a scripting language. In: POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 377–388 (2010)

Appendix

1 Conventions

We omit the blame label ℓ from coercions and values when it is not significant to simplify our presentation. We define some additional terms and notational conventions below. Some of these are repeated from the main content of the paper for convenience.

Definition 1. (direct superclass, superclass and proper superclass)

Assume class $C<\overline{X}>$ extends $D<\overline{V}>$ { ... }. Now D is the direct superclass of C (or, equivalently, C directly inherits D). E is a superclass of C if E = C, or if E is a superclass of the direct superclass of C. E is a proper superclass of C if E is a superclass of C and E $\neq$ C.

Definition 2. (subclass and proper subclass)

If D is a superclass of C, then C is a subclass of D. If D is a proper superclass of C, then C is a proper subclass of D.

Definition 3. (inheritance hierarchy depth between classes)

The inheritance hierarchy depth between classes C and D is inductively defined.

If C = D, the depth between C and D is 0.

Otherwise, if E is the direct superclass of C, the depth between C and D is 1 + the depth between E and D.

Definition 4. (alternative notation for $map_{\uparrow}$ and $map_{\downarrow}$)

It is occasionally convenient to use an alternative notation for $map_{\uparrow}$ and $map_{\downarrow}$:

1. $map_{\uparrow}(C<\overline{T}>, D) = D<\overline{S}>$ may be written as $\overline{T}:C/D = \overline{S}$.
2. $map_{\downarrow}(D<\overline{S}>, C, \overline{T}) = C<\overline{U}>$ may be written as $\overline{S}:D/C<\overline{T}> = \overline{U}$.

Definition 5. (the depth of a type)

We use depth(T) for the depth of a type. The function depth is defined recursively below.

Simple types:

depth(dyn) = 1
depth(Object) = 1
depth(X) = 1 (type variable)

Case $C<\overline{T}>$:

Assume class $C<\overline{X}>$ extends $D<\overline{V}>$ { ... }.

Now $depth(C<\overline{T}>) =$
$\quad max(1 + max(depth(T_i) \text{ for } i \text{ in } 1..\#(\overline{T})),$
$\qquad depth(map_{\uparrow}(C<\overline{T}>, D)))$.

The above assumes that max(X) = 0 if X is empty.

Definition 6. (the depth of a list of types)

We define $depth(\overline{T})$ as a shorthand for $max(depth(T_i))$, i in $1..\#(\overline{T})$.

2 Properties of $map_{\downarrow}$ and $map_{\uparrow}$ and depth

Lemma 1. $C<\overline{T}> <: map_{\uparrow}(C<\overline{T}>, D)$.

Proof.

Induction on inheritance hierarchy depth between C and D.

Base case (C = D):

Trivial.

Induction step (depth > 0):

Assume C inherits E directly.

It is easy to see that $C<\overline{T}> <: map_{\uparrow}(C<\overline{T}>, E)$, from definition of subtyping and $map_{\uparrow}$.

Apply induction hypothesis on $map_{\uparrow}(C<\overline{T}>, E) <: map_{\uparrow}(C<\overline{T}>, D)$, and then use transitivity of subtyping. ■

Lemma 2. $map_{\uparrow}(C<\overline{T}>, D) <: map_{\uparrow}(C<\overline{T}>, E)$ if D is a subclass of E.

Proof.

First we notice that $C<\overline{T}> <: map_{\uparrow}(C<\overline{T}>, D)$ (by lemma 1).

Let $D<\overline{U}> = map_{\uparrow}(C<\overline{T}>, D)$. Now let $E<\overline{V}> = map_{\uparrow}(D<\overline{U}>, E)$; from the definition of $map_{\uparrow}$ it is easy to see that $E<\overline{V}> = map_{\uparrow}(C<\overline{T}>, E)$.

Using lemma 1 we get $D<\overline{U}> <: E<\overline{V}>$, i.e. $map_{\uparrow}(C<\overline{T}>, D) <: map_{\uparrow}(C<\overline{T}>, E)$. ■

Lemma 3. If $C<\overline{T}> <: D<\overline{S}>$, then $map_{\uparrow}(C<\overline{T}>, D) = D<\overline{S}>$.

Proof.

Use induction on inheritance hierarchy depth between C and D.

(An alternative method would involve the formulation of an algorithmic subtyping relation. We could prove its equivalency to our definition of subtyping.)

Base case C = D:

Trivial.

Base case, C directly inherits D:

Trivial, from definition of <: and $map_{\uparrow}$.

Induction step (inheritance depth > 1):

Assume class $C<\overline{X}>$ extends $E<\overline{U}>$ { ... }.

Based on definition of <:, if $C<\overline{T}> <: D<\overline{S}>$, there must be $E<\overline{V}>$ such that $C<\overline{T}> <: E<\overline{V}> <: D<\overline{S}>$.

We can easily see that $E<\overline{V}> = map_{\uparrow}(C<\overline{T}>, E)$, from definition of $<:$ and $map_{\uparrow}$.

Now by induction hypothesis, $map_{\uparrow}(E<\overline{V}>, D) = D<\overline{S}>$, and from combining the above and the definition of $map_{\uparrow}$ we get $map_{\uparrow}(C<\overline{T}>, D) = map_{\uparrow}(map_{\uparrow}(C<\overline{T}>, E), D) = D<\overline{S}>$. ■

<u>Lemma 4</u>. If $depth(C<\overline{T}>) = n$, then $depth(T_i) < n$ for all i in $1..\#(\overline{T})$.

<u>Proof</u>.

By definition of depth, $depth(C<\overline{T}>) \geq 1 + max(depth(T_i))$
$\Rightarrow depth(T_i) \leq depth(C<\overline{T}>) - 1 \Rightarrow depth(T_i) < depth(C<\overline{T}>)$. ■

<u>Lemma 5</u>. $depth(map_{\uparrow}(C<\overline{T}>, D)) \leq depth(C<\overline{T}>)$.

<u>Proof</u>.

By induction on class hierarchy depth between C and D.

Base case C = D:

Trivial.

Induction step:

Assume class $C<\overline{X}>$ extends $E<\overline{V}>$ { ... }.

Now by definition of depth, it is easy to see that $depth(map_{\uparrow}(C<\overline{T}>, E)) \leq depth(C<\overline{T}>)$.

Use induction hypothesis $\Rightarrow depth(map_{\uparrow}(map_{\uparrow}(C<\overline{T}>, E), D)) \leq depth(map_{\uparrow}(C<\overline{T}>, E)) \Rightarrow depth(map_{\uparrow}(C<\overline{T}>, D)) \leq depth(C<\overline{T}>)$
(by definition of $map_{\uparrow}$ and since $depth(map_{\uparrow}(C<\overline{T}>, E)) \leq depth(C<\overline{T}>)$). ■

<u>Lemma 6</u>. $depth(map_{\downarrow}(D<\overline{S}>, C, \overline{T})) \leq max(depth(C<\overline{T}>), depth(D<\overline{S}>))$ if $map_{\downarrow}(D<\overline{S}>, C, \overline{T})$ exists.

<u>Proof</u>.

Induction on inheritance hierarchy depth between C and D.

Base case C = D:

Trivial.

Base case C directly inherits D:

Now, by definition of $map_{\downarrow}$:

class $C<\overline{X}>$ extends $D<\overline{V}>$ { ... }
$[\overline{U}/\overline{X}]D<\overline{V}> = D<\overline{S}>$
if X_i not in $ftv(\overline{V})$ then $U_i = T_i$

and now

$map_{\downarrow}(D<\overline{S}>, C, \overline{T}) = C<\overline{U}>$.

Now consider each U_i separately. If for each U_i, $depth(U_i) < max(depth(C<\overline{T}>), depth(D<\overline{S}>))$, then our desired conculusion follows.

If X_i not in $ftv(\overline{V})$, then $U_i = T_i$. Clearly $depth(T_i) < depth(C<\overline{T}>)$.

Otherwise, U_i must have been substituted in $D<\overline{S}>$; clearly $depth(U_i) < depth(D<\overline{S}>)$. But now we have shown that for each U_i, $depth(U_i) < max(depth(C<\overline{T}>), depth(D<\overline{S}>))$
$\Rightarrow depth(C<\overline{U}>) = depth(map_{\downarrow}(D<\overline{S}>, C, \overline{T})) \leq max(depth(C<\overline{T}>), depth(D<\overline{S}>))$.

Induction step (depth > 1 between C and D):

By definition of $map_{\downarrow}$,

$map_{\downarrow}(D<\overline{S}>, C, \overline{T}) = map_{\downarrow}(map_{\downarrow}(D<\overline{S}>, E, \overline{T}:C/E), C, \overline{T})$ for some E.

Now by induction hypothesis and $map_{\uparrow}$ depth lemma 5,

$depth(map_{\downarrow}(D<\overline{S}>, E, \overline{T}:C/E)) \leq max(depth(C<\overline{T}>), depth(D<\overline{S}>))$. (1)

Use induction hypothesis again, to get

$depth(map_{\downarrow}(map_{\downarrow}(D<\overline{S}>, E, \overline{T}:D/E), C, \overline{T})) \leq max(depth(C<\overline{T}>), depth(map_{\downarrow}(D<\overline{S}>, E, \overline{T}:D/E))) \leq max(depth(C<\overline{T}>), depth(D<\overline{S}>))$ (use equation 1). ■

<u>Lemma 7</u>. If $C<\overline{T}> = map_{\downarrow}(D<\overline{S}>, C, \overline{V})$ exists, it is unique.

<u>Proof</u>.

Proof by contradiction using induction on class hierarchy depth between C and D.

Assume $C<\overline{U}> = map_{\downarrow}(D<\overline{S}>, C)$ and $C<\overline{U}> \neq C<\overline{T}>$.

Base case C = D:

Trivial.

Induction step:

Assume class $C<\overline{X}>$ extends $E<\overline{R}>$ { ... }.

Let $E<\overline{W}> = map_{\downarrow}(D<\overline{S}>, E, \overline{V}:C/E)$ (this exists by lemma 9).
By the induction hypothesis, $E<\overline{W}>$ is unique, so we can reformulate the assumptions: $C<\overline{T}> = map_{\downarrow}(E<\overline{W}>, C, \overline{V})$ (similarly for $C<\overline{U}>$).

By definition of of $map_{\downarrow}$:

$[\overline{U}/\overline{X}]E<\overline{R}> = E<\overline{W}>$, some $U_i = V_i$
$[\overline{T}/\overline{X}]E<\overline{R}> = E<\overline{W}>$, some $T_i = V_i$

If X_i not in $ftv(\overline{R})$, then both $U_i = V_i$ and $T_i = V_i$. If $\overline{U}$ and $\overline{T}$ are different, the difference must be in T_i/U_i so that X_i in $ftv(\overline{V})$.

By lemma 31, if $[\overline{U}/\overline{X}]Q = [\overline{T}/\overline{X}]Q$, then $U_i = T_i$ for each i such that X_i in $ftv(Q)$. Then select $Q = E<\overline{R}>$, and we have found a contradiction. ■

<u>Lemma 8</u>. $map_{\uparrow}(map_{\downarrow}(C<\overline{T}>, D, \overline{S}), E) = map_{\downarrow}(C<\overline{T}>, E, \overline{S}:D/E)$ if E is subclass of C and D is subclass of E (assuming $map_{\downarrow}$ exists).

<u>Proof</u>.

It is easy to see that this is equivalent to proving $map_{\uparrow}(map_{\downarrow}(C<\overline{T}>, D, \overline{S}), C) = C<\overline{T}>$ (by definition of $map_{\downarrow}$).

Prove for D as direct subtype of C; the rest follows easily using induction (details omitted).

Assume class $D<\overline{X}>$ extends $C<\overline{V}>$ { ... }.

If $D<\overline{U}> = map_{\downarrow}(C<\overline{T}>, D, \overline{S})$ then $[\overline{U}/\overline{X}]C<\overline{V}> = C<\overline{T}>$ (by definition of $map_{\downarrow}$; the additional constraints for $\overline{U}$ in the definition of $map_{\downarrow}$ do not affect this proof).

Now $map_{\uparrow}(D<\overline{U}>, C) = [\overline{U}/\overline{X}]C<\overline{V}> = C<\overline{T}>$ as expected. ■

Lemma 9. If $map_{\downarrow}(C<\overline{T}>, E, \overline{S})$ exists, and D is a subclass C and a superclass of E, then $map_{\downarrow}(C<\overline{T}>, D, \overline{S}:E/D)$ and $map_{\downarrow}(map_{\downarrow}(C<\overline{T}>, D, \overline{S}:E/D), E, \overline{S})$ exist.

Proof.

By induction on inheritance hierarchy depth between E and C. Easy; use definition of $map_{\downarrow}$.

Lemma 10. $map_{\downarrow}(map_{\uparrow}(C<\overline{T}>, D), C, \overline{T}) = C<\overline{T}>$.

Proof.

Proof by induction on class hierarchy depth between C and D.

Base case (C = D):

Trivial.

Base case (C directly inherits D):

Assume class $C<\overline{X}>$ extends $D<\overline{V}>$ { ... }.

$D<\overline{S}> = map_{\uparrow}(C<\overline{T}>, D)$
$\Rightarrow D<\overline{S}> = [\overline{T}/\overline{X}]D<\overline{V}>$ (by definition of $map_{\uparrow}$). (1)

Now by definition of $map_{\downarrow}$; $C<\overline{W}> = map_{\downarrow}(...)$:

$[\overline{W}/\overline{X}]D<\overline{V}> = D<\overline{S}>$, (2)
if X_i not in $ftv(\overline{V})$ then $W_i = T_i$.

If X_i not in $ftv(\overline{V})$, $R_i = T_i$ trivially. Now we need only to focus on X_i in $ftv(\overline{V})$.

Combine equations (1) and (2) to get

$[\overline{T}/\overline{X}]D<\overline{V}> = [\overline{W}/\overline{X}]D<\overline{V}>$.

Now by lemma 31, $W_i = T_i$ for X_i in $ftv(\overline{V})$.

Now for X_i in $ftv(\overline{V})$, $W_i = T_i$, which finishes this case.

Induction step:

There must be a class E in hierarchy between C and D.

It is easy to see that $map_{\downarrow}(map_{\uparrow}(C<\overline{T}>, D), C, \overline{T})$ can be written as

$map_{\downarrow}(map_{\downarrow}(map_{\uparrow}(map_{\uparrow}(C<\overline{T}>, E), D),$
$E, \overline{T}:C/E),$
$C, \overline{T})$.

Now use induction hypothesis on

$map_{\downarrow}(map_{\uparrow}(E<\overline{U}>, D), E, \overline{U})$ where $E<\overline{U}> = map_{\uparrow}(C<\overline{T}>, E)$

and replace it with $E<\overline{U}>$ in the original equation to get

$map_{\downarrow}(E<\overline{U}>, C, \overline{T}) = map_{\downarrow}(map_{\uparrow}(C<\overline{T}>, E), C, \overline{T})$.

Now use induction hypothesis again to get the desired conclusion:

$map_{\downarrow}(map_{\uparrow}(C<\overline{T}>, E), C, \overline{T}) = C<\overline{T}>$. ■

3 Properties of $\prec$, $\prec_{co}$ and $\prec_{=}$

Lemma 11. If $C<\overline{T}> \prec D<\overline{S}>$ or $C<\overline{T}> \prec_{co} D<\overline{S}>$, then C is a subclass of D.

Proof.

Easy, by definition of $\prec$, $\prec_{co}$ and $map_{\uparrow}$.

Lemma 12. If T <: S, then $T \prec S$ and $T \prec_{co} S$.

Proof.

Use properties of $map_{\uparrow}$.

Case dyn <: dyn:

Trivial.

Case $C<\overline{T}> <: D<\overline{S}>$:

Now $map_{\uparrow}(C<\overline{T}>, D) = D<\overline{S}>$ (by lemma 3).

Both $\prec$ and $\prec_{co}$ immediately follow.

Lemma 13. If $T \prec S$ then $T \prec_{co} S$.

Proof.

Proof by induction on maximum depth of T and S.

Base cases:

Trivial.

Induction step:

Case $C<\overline{T}> \prec$ dyn:

Trivial.

Case $C<\overline{T}> \prec D<\overline{S}>$:

Use induction hypothesis on $\overline{T}$ and $\overline{S}$; the result immediately follows.

Lemma 14. $T \prec S$ and $T \sim S$ if and only if $T \prec_{=} S$.

Proof.

Induction on maximum depth of T and S.

"$\Leftarrow$"

Base cases:

Case dyn, dyn:
Trivial.

Case C, C:
Trivial.
Case C, D:
Trivial ($C \neq D \Rightarrow$ not $C \prec_{=} S$).
Case dyn, C:
Trivial (not $dyn \prec_{=} C$).
Case C, dyn:
Trivial.

Induction step:

Case $C<\overline{T}>, D<\overline{S}>$ ($C \neq D$):
Impossible.
Case $C<\overline{T}>$, dyn:
Trivial.
Case dyn, $C<\overline{T}>$:
Trivial.
Case $C<\overline{T}>, C<\overline{S}>$:
From induction hypothesis and definition of $\prec_{=}$.

"$\Rightarrow$"

Base cases:

Case dyn, dyn:
Trivial.
Case C, C:
Trivial.
Case C, D:
Trivial ($C \neq D \Rightarrow$ not $C \sim D$).
Case dyn, C:
Trivial (not $dyn \prec C$).
Case C, dyn:
Trivial.

Induction step:

Case $C<\overline{T}>, D<\overline{S}>$:
Trivial ($C \neq D$).
Case $C<\overline{T}>$, dyn:
Trivial.
Case dyn, $C<\overline{T}>$:
Trivial.
Case $C<\overline{T}>, C<\overline{S}>$:
From induction hypothesis and definition of $\prec$. ■

Lemma 15. If $C<\overline{T}> \prec_{=} C<\overline{S}>$ and $\overline{S}$ has no dyn then $\overline{T} = \overline{S}$.

Proof.

Immediate from definition of $\prec_{=}$.

Lemma 16. (transitivity of $\prec_{co}$) If $U \prec_{co} T$ and $T \prec_{co} S$ then $U \prec_{co} S$.

Proof.

Induction on depth of S.

Base case:

Case $dyn \prec_{co} dyn \prec_{co} dyn$:

Trivial.

Case $C \prec_{co} dyn \prec_{co} dyn$:

Trivial.

Case $C \prec_{co} D \prec_{co} dyn$:

Trivial.

Case $C \prec_{co} D \prec_{co} E$:

Trivial.

Case $C<\overline{T}> \prec_{co} D<\overline{S}> \prec_{co} E$:

$map_{\uparrow}(C<\overline{T}>, E) = E$ (by definition of $map_{\uparrow}$).
(C must be a subclass of D, and D must be a subclass of E $\Rightarrow$
C must be a subclass of E)
$\Rightarrow C<\overline{T}> \prec_{co} E$.

Induction step:

Trivial if U = T or T = S.

The only interesting case is if everything is an instance type.

$C<\overline{T}> \prec_{co} D<\overline{S}>, D<\overline{S}> \prec_{co} E<\overline{U}>$.

$E<\overline{T}'> = map_{\uparrow}(C<\overline{T}>, E)$.
$E<\overline{S}'> = map_{\uparrow}(D<\overline{S}>, E)$.

$E<\overline{T}'> \prec_{co} E<\overline{S}'>$ (by lemma 22) and $E<\overline{S}'> \prec_{co} E<\overline{U}>$ (by lemma 22).
By definition of $\prec_{co}$, $\overline{T}' \prec_{co} \overline{S}'$ and $\overline{S}' \prec_{co} \overline{U}$.

Use induction hypothesis $\Rightarrow \overline{T}' \prec_{co} \overline{U}$.

$\Rightarrow C<\overline{T}> \prec_{co} E<\overline{U}>$ (by definition of $\prec_{co}$). ■

Lemma 17. (transitivity of $\prec$) If $U \prec T$ and $T \prec S$ then $U \prec S$.

See also lemma 16 (transitivity of $\prec_{co}$) for more details.

Proof.

Induction on depth of S.

Base case:

Case $dyn \prec dyn \prec dyn$:

Trivial.

(Omit trivial cases that are similar to lemma 16.)

Case $C<\overline{T}> \prec D<\overline{S}> \prec E$:

$map_{\uparrow}(C<\overline{T}>, E) = E$ (by definition of $map_{\uparrow}$)
$\Rightarrow C<\overline{T}> \prec E$.

Induction step:

Now we have $C<\overline{T}> \prec D<\overline{S}>, D<\overline{S}> \prec E<\overline{U}>$, and we show that $C<\overline{T}> \prec E<\overline{U}>$.

$E<\overline{T}'> = map_{\uparrow}(C<\overline{T}>, E)$.
$E<\overline{S}'> = map_{\uparrow}(D<\overline{S}>, E)$.

$C<\overline{T}> \prec D<\overline{S}> \Rightarrow E<\overline{T}'> \prec E<S'>$ (by lemma 21).
$D<\overline{S}> \prec E<\overline{U}> \Rightarrow E<\overline{S}'> \prec E<\overline{U}'>$ (by lemma 21).

Now by definition of $\prec$, $\overline{T}' \prec_= \overline{S}'$ and $\overline{S}' \prec_= \overline{U}'$.

$\Rightarrow \overline{T}' \prec_= \overline{U}'$ (by induction hypothesis).

Now $C<\overline{T}> \prec E<\overline{U}>$ (use definition of $\prec$). ■

Lemma 18. (transitivity of $\prec_=$) If $U \prec_= T$ and $T \prec_= S$ then $U \prec_= S$.

Proof.

Induction on depth of S.

Base cases:

Case $dyn \prec_= dyn \prec_= dyn$:
Trivial.
Case $C<\overline{T}> \prec_= dyn \prec_= dyn$:
Trivial.
Case $C<\overline{T}> \prec_= C<\overline{S}> \prec_= dyn$:
Trivial.
Case $C \prec_= C \prec_= C$:
Trivial.

Induction step:

Case $C<\overline{T}> \prec_= C<\overline{S}> \prec_= C<\overline{U}>$:

We get $\overline{T} \prec_= \overline{S} \prec_= \overline{U}$; from induction hypothesis $\overline{T} \prec_= \overline{U}$
$\Rightarrow C<\overline{T}> \prec_= C<\overline{U}>$. ■

Lemma 19. If $U <: T$ and $T \prec_{co} S$, then $U \prec_{co} T$.

Proof.

By lemma 12, $U <: T \Rightarrow U \prec_{co} T$.
By lemma 16, $U \prec_{co} T$ and $T \prec_{co} S \Rightarrow U \prec_{co} S$. ■

Lemma 20. If $C<\overline{T}> \prec_{co} D<\overline{S}>$, and C inherits E, and $E<\overline{U}> = map_\uparrow(C<\overline{T}>, E)$, then $E<\overline{U}> \prec_{co} D<\overline{S}>$.

Proof.

If $C<T> \prec_{co} D<\overline{S}>$, then we can find (by definition of $\prec_{co}$) $D<\overline{U}> = map_\uparrow(C<\overline{T}>, D)$, and $\overline{U} \prec_{co} \overline{S} \Rightarrow D<\overline{U}> \prec_{co} D<\overline{S}>$ (by definition of $\prec_{co}$).

Also $map_\uparrow(C<\overline{T}>, E) <: D<\overline{U}>$ (by lemma 2).

Using the above and lemma 19 $\Rightarrow map_\uparrow(C<\overline{T}>, E) \prec_{co} D<\overline{S}>$. ■

Lemma 21. If $C<\overline{T}> \prec D<\overline{S}>$, then $map_\uparrow(C<\overline{T}>, E) \prec map_\uparrow(D<\overline{S}>, E)$.

See also lemma 22.

Proof.

Case D = E:

Show $C<\overline{T}> \prec D<\overline{S}> \Rightarrow map_\uparrow(C<\overline{T}>, D) \prec D<\overline{S}>$.

Now $map_\uparrow(C<\overline{T}>, D) = D<\overline{W}>$, $\overline{W} \prec_= \overline{S}$ (by definition of $\prec$ and lemma 14).

$\Rightarrow map_\uparrow(C<\overline{T}>, D) \prec D<\overline{S}>$ (by definition of $\prec$ and lemma 14).

Case $D \neq E$:

First, by above, $D<\overline{U}> = map_\uparrow(C<\overline{T}>, D) \prec D<\overline{S}>$.

Now we show that if $D<\overline{U}> \prec D<\overline{S}>$, then $map_\uparrow(D<\overline{U}>, E) \prec map_\uparrow(D<\overline{S}>, E)$.

Use induction on depth of inheritance hierarchy between D and E.

Base case (E is direct supertype of D):

Assume class $D<\overline{X}>$ extends $E<\overline{V}>$ { ... }.

Since $D<\overline{U}> \prec D<\overline{S}>$, $\overline{U} \prec_= \overline{S}$ (by definition of $\prec$).

Now show that $D<\overline{U}> \prec map_\uparrow(D<\overline{S}>, E) = E<\overline{W}>$:

This is equivalent to showing that $D<\overline{U}> \prec [\overline{S}/\overline{X}]E<\overline{V}>$ (by definition of $map_\uparrow$).

Now by definition of $\prec$ we need to show that if $E<\overline{U}'> = map_\uparrow(D<\overline{U}>, E)$ then $\overline{U}' \prec_= \overline{W}$
$\Leftrightarrow [\overline{U}/\overline{X}]\overline{V} \prec_= [\overline{S}/\overline{X}]\overline{V}$ (by definition of $map_\uparrow$).

Since substitution preserves $\prec_=$ (by lemma 27) and $\overline{U} \prec_= \overline{S}$, the above is true, i.e. $D<\overline{U}> \prec map_\uparrow(D<\overline{S}>, E)$.

Now we can use the case D = E above to show that $map_\uparrow(D<\overline{U}>, E) \prec map_\uparrow(D<\overline{S}>, E)$.

Induction step:

Trivial, just go up through the inheritance hierarchy. ■

Lemma 22. If $C<\overline{T}> \prec_{co} D<\overline{S}>$, then $map_\uparrow(C<\overline{T}>, E) \prec_{co} map_\uparrow(D<\overline{S}>, E)$.

Proof.

Case D = E:

Use induction on depth of inheritance hierarchy between C and D with lemma 20.

Case $D \neq E$:

First, by above, $D<\overline{U}> = map_\uparrow(C<\overline{T}>, D) \prec_{co} D<\overline{S}>$.

Now we need to show that if $D<\overline{U}> \prec_{co} D<\overline{S}>$, then $map_\uparrow(D<\overline{U}>, E) \prec_{co} map_\uparrow(D<\overline{S}>, E)$.

Use induction on depth of inheritance hierarchy between D and E.

Base case (E is direct supertype of D):

First show this:
$D<\overline{U}> \prec_{co} map_\uparrow(D<\overline{S}>, E)$.
$D<\overline{U}> \prec_{co} [\overline{S}/\overline{X}]E<\overline{V}>$ (assume class $D<\overline{X}>$ extends $E<\overline{V}>$ { ... }).

Now use definition of $\prec_{co}$ (premise):

$E<\overline{U}'> = map_\uparrow(D<\overline{U}>, E)$, $\overline{U}' \prec_{co} [\overline{S}/\overline{X}]\overline{V}$
$\Rightarrow E<\overline{U}'> = [\overline{U}/\overline{X}]E<\overline{V}>$, $[\overline{U}/\overline{X}]\overline{V} \prec_{co} [\overline{S}/\overline{X}]\overline{V}$.

Since substitution preserves $\prec_{co}$ (lemma 26), we can get to the desired conclusion:

$$\mathtt{D<\overline{U}>} \prec_{co} \mathtt{map}_\uparrow(\mathtt{D<\overline{S}>, E}).$$

Now we can use the case D = E above to show that $\mathtt{map}_\uparrow(\mathtt{D<\overline{U}>, E}) \prec_{co} \mathtt{map}_\uparrow(\mathtt{D<\overline{S}>, E})$.

Induction step:

Trivial, just go up through the inheritance hierarachy. ■

<u>Lemma 23</u>. $\mathtt{map}_\downarrow(\mathtt{C<\overline{T}>, D, \overline{S}}) \prec_{co} \mathtt{C<\overline{T}>}$, if $\mathtt{map}_\downarrow(\mathtt{C<\overline{T}>, D, \overline{S}})$ exists.

<u>Proof</u>.

By definition of $\prec_{co}$:

$\mathtt{map}_\downarrow(\mathtt{C<\overline{T}>, D, \overline{S}}) \prec_{co} \mathtt{C<\overline{T}>}$
$\Leftrightarrow \mathtt{C<\overline{U}>} = \mathtt{map}_\uparrow(\mathtt{map}_\downarrow(\mathtt{C<\overline{T}>, D, \overline{S}}), \mathtt{C})$ and $\overline{\mathtt{U}} \prec_{co} \overline{\mathtt{T}}$.

But by lemma 8, $\mathtt{C<\overline{U}>} = \mathtt{C<\overline{T}>}$, and now trivially $\overline{\mathtt{U}} \prec_{co} \overline{\mathtt{T}}$. ■

<u>Lemma 24</u>. If $\mathtt{D<\overline{S}>} \prec_{co} \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, D})$, then $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}}) \prec_{co} \mathtt{C<\overline{T}>}$, whenever $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}})$ exists.

<u>Proof</u>.

Proof by induction on inheritance hierarchy depth between C and D.

Base case C = D:

Trivial.

Induction step:

Assume class $\mathtt{E<\overline{X}>}$ extends $\mathtt{D<\overline{V}>}$ { ... }.

Let $\mathtt{E<\overline{U}>} = \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, E})$.

Now $\mathtt{D<\overline{S}>} \prec_{co} \mathtt{map}_\uparrow(\mathtt{E<\overline{U}>, D})$ (by definition of $\mathtt{map}_\uparrow$ and the premise $\mathtt{D<\overline{S}>} \prec_{co} \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, D})$).

By lemma 9, $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, E, \overline{T}:C/E})$ and $\mathtt{map}_\downarrow(\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, E, \overline{T}:C/E}), \mathtt{C, \overline{T}})$ exist since $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C})$ exists by lemma assumptions.

By lemma 25, $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, E, \overline{T}:C/E}) \prec_{co} \mathtt{E<\overline{U}>}$.

We have shown that $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, E, \overline{T}:C/E}) \prec_{co} \mathtt{E<\overline{U}>} = \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, E})$.
Now, by induction hypothesis,
$\mathtt{map}_\downarrow(\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, E, \overline{T}:C/E}), \mathtt{C, \overline{T}}) \prec_{co} \mathtt{C<\overline{T}>}$, but this is equivalent to $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}}) \prec_{co} \mathtt{C<\overline{T}>}$ (by definition of $\mathtt{map}_\downarrow$). ■

<u>Lemma 25</u>. If class $\mathtt{C<\overline{X}>}$ extends $\mathtt{D<\overline{V}>}$ and $\mathtt{D<\overline{S}>} \prec_{co} \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, D})$, then $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}}) \prec_{co} \mathtt{C<\overline{T}>}$, whenever $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}})$ exists.

Notes.

The final conditition is not trivial. Below we give an example where it makes a difference.

Assume class $\mathtt{C<X>}$ extends $\mathtt{D<X, X>}$ { ... }.

Assume $\mathtt{C<\overline{T}>} = \mathtt{C<dyn>}$ and $\mathtt{D<\overline{S}>} = \mathtt{D<A, dyn>}$. Now $\mathtt{map}_\uparrow(\mathtt{C<dyn>, D}) = \mathtt{D<dyn, dyn>}$.

Note that $\mathtt{D<A, dyn>} \prec_{co} \mathtt{D<dyn, dyn>}$ but $\mathtt{map}_\downarrow(\mathtt{D<A, dyn>, C})$ does not exist.

<u>Proof</u>.

Let $\mathtt{D<\overline{U}>} = \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, D}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{V}>}$ (by definition of $\mathtt{map}_\uparrow$).

Since $\mathtt{D<\overline{S}>} \prec_{co} \mathtt{map}_\uparrow(\mathtt{C<\overline{T}>, D})$, $\overline{\mathtt{S}} \prec_{co} \overline{\mathtt{U}}$ (by definition of $\prec_{co}$).

If $\mathtt{C<\overline{W}>} = \mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}})$, then $[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{D<\overline{V}>} = \mathtt{D<\overline{S}>}$ and some $\mathtt{W_i} = \mathtt{T_i}$ (by definition of $\mathtt{map}_\downarrow$). Note that we originally assumed that this exists.

Now by the above:

$\overline{\mathtt{S}} = [\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}$, $\mathtt{W_i} = \mathtt{T_i}$ if ...
$\overline{\mathtt{U}} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}$

$\overline{\mathtt{S}} \prec_{co} \overline{\mathtt{U}}$

We need to show that $\overline{\mathtt{W}} \prec_{co} \overline{\mathtt{T}}$.

For $\mathtt{W_i} = \mathtt{T_i}$ this is trivial (by definition of $\prec_{co}$).

Now focus on the rest of the cases.

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \prec_{co} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}$

This is equivalent to $[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{D<\overline{V}>} \prec_{co} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{V}>}$. By lemma 32, this implies that $\mathtt{W_i} \prec_{co} \mathtt{T_i}$ for all $\mathtt{X_i}$ in $\mathtt{ftv(D<\overline{V}>)}$.

Combining the above result and the previous result for $\mathtt{W_i} = \mathtt{T_i}$, we get the desired result, if $\mathtt{map}_\downarrow(\mathtt{D<\overline{S}>, C, \overline{T}})$ exists.

4 Substitution lemmas

<u>Lemma 26</u>. If $\overline{\mathtt{T}} \prec_{co} \overline{\mathtt{S}}$ then $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{V} \prec_{co} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{V}$.

<u>Proof</u>.

By induction on depth of V.

Base cases:

Case X: Trivial.
Case C: Trivial.
Case dyn: Trivial.

Induction step:

Case $\mathtt{C<\overline{T}>}$:

Immediately follows from induction hypothesis (using definition of $\prec_{co}$).

<u>Lemma 27</u>. If $\overline{\mathtt{T}} \prec_{=} \overline{\mathtt{S}}$ then $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{V} \prec_{=} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{V}$, i.e. if $\overline{\mathtt{T}} \prec \overline{\mathtt{S}}$ and $\overline{\mathtt{T}} \sim \overline{\mathtt{S}}$ then $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{V} \prec_{=} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{V}$ (by lemma 14).

See also lemma 26.

<u>Proof</u>.

By induction on depth of V.

Base cases:

Case X: Trivial.
Case C: Trivial.
Case dyn: Trivial.

Induction step:

Case $C<\overline{U}>$:

Immediately follows from induction hypothesis (by definition of $\prec_=$).

Lemma 28. $[\overline{T}/\overline{X}]\mathrm{mtype}(m, C<\overline{S}>) = \mathrm{mtype}(m, [\overline{T}/\overline{X}]C<\overline{S}>)$.

Proof.

By induction on inheritance hierarchy depth between C and the nearest superclass that defines m.

Base case (m defined in C):

Assume m has signature $\overline{U} \to U_0$.

Second: $[\overline{T}/\overline{X}]\mathrm{mtype}(m, C<\overline{S}>) = [\overline{T}/\overline{X}][\overline{S}/\overline{X}](\overline{U} \to U_0)$.
First: $\mathrm{mtype}(m, [\overline{T}/\overline{X}]C<\overline{S}>) = \mathrm{mtype}(m, C<[\overline{T}/\overline{X}]\overline{S}>) = [[\overline{T}/\overline{X}]\overline{S}/\overline{X}](\overline{U} \to U_0)$.

$[\overline{T}/\overline{X}][\overline{S}/\overline{X}]U = [([\overline{T}/\overline{X}]\overline{S})/\overline{X}]U$ by lemma 29.

Case m inherited:

Consider for a single inheritance step; use induction hypothesis for a deeper inheritance hierarchy (details omitted).

Assume class $C<\overline{X}>$ extends $D<\overline{V}>$ { ... }.

Second case:
$[\overline{T}/\overline{X}]\mathrm{mtype}(m, C<\overline{S}>)$
$= [\overline{T}/\overline{X}]\mathrm{mtype}(m, [\overline{S}/\overline{X}]D<\overline{V}>)$
$= [\overline{T}/\overline{X}]\mathrm{mtype}(m, D<[\overline{S}/\overline{X}]\overline{V}>)$
$= [\overline{T}/\overline{X}][[\overline{S}/\overline{X}]\overline{V}/\overline{Y}](\overline{U} \to U_0)$.
First case:
$\mathrm{mtype}(m, [\overline{T}/\overline{X}]C<\overline{S}>)$
$= \mathrm{mtype}(m, [[\overline{T}/\overline{X}]\overline{S}/\overline{X}]D<\overline{V}>)$
$= \mathrm{mtype}(m, D<[[\overline{T}/\overline{X}]\overline{S}/\overline{X}]\overline{V}>)$
$= [[[\overline{T}/\overline{X}]\overline{S}/\overline{X}]/\overline{Y}](\overline{U} \to U_0)$.

Now use lemma 29 twice to get the equality.

Lemma 29. $[\overline{T}/\overline{X}][\overline{S}/\overline{X}]U = [([\overline{T}/\overline{X}])\overline{S}/\overline{X}]U$.

Proof.

Induction on depth of U.

Base cases:

Case C:
Trivial.

Case dyn:
Trivial.

Case X:
In the first case, we replace $X \Rightarrow S_i$; then we replace S_i with $[\overline{T}/\overline{X}]S_i$.

In the second case, we substitute $[\overline{T}/\overline{X}]$ for S_i, then replace X with $[\overline{T}/\overline{X}]S_i$.

(Or trivial, if X is not substituted.)

Induction step:

Case $C<\overline{V}>$:

Easy; use induction hypothesis.

Lemma 30. $\mathrm{fields}([\overline{S}/\overline{X}]T) = [\overline{S}/\overline{X}]\mathrm{fields}(T)$.

Proof.

Similar to lemma 28.

Lemma 31. If $[\overline{T}/\overline{X}]U = [\overline{S}/\overline{X}]U$, then for each i such that X_i in ftv(U), $T_i = S_i$.

Proof.

By induction on depth of U.

Base cases:

Case dyn:

Trivial.

Case C:

Trivial.

Case X_i:

Now ftv(U) only has X_i. $[\overline{T}/\overline{X}]X_i = T_i$ and $[\overline{S}/\overline{X}]X_i = S_i$; but now T_i must be equal to S_i, since $[\overline{T}/\overline{X}]X_i = [\overline{S}/\overline{X}]X_i$ by assumptions.

Induction step:

Case $C<\overline{V}>$:

Just use induction hypothesis for $\overline{V}$; then note that $\mathrm{ftv}(C<\overline{V}>) = \mathrm{union}(\mathrm{ftv}(V_i))$. Combine these, and the desired conclusion follows. ■

Lemma 32. If $[\overline{W}/\overline{X}]S \prec_{co} [\overline{T}/\overline{X}]S$, then $W_i \prec_{co} T_i$ for all X_i in ftv(S).

Proof.

Proof by induction on depth of S.

Base cases:

Case dyn and C:

Trivial.

Case X_i:

We get immediately $W_i \prec_{co} T_i$; also X_i in ftv(S).

Induction step:

$D<\overline{U}>$; apply induction step to each U. But $ftv(\overline{U}) = ftv(D<\overline{U}>)$ so from this we get the result for $D<\overline{U}>$

5 Properties of mtype

Definition 7. $(\overline{U} \to U_0) \prec (\overline{V} \to V_0)$ if and only if $\overline{U} \prec \overline{V}$ and $U_0 \prec V_0$. (And similarly for $\prec_{co}$ and $\prec_{=}$).

Lemma 33. If $C<\overline{T}> \prec C<\overline{S}>$, then $mtype(m, C<\overline{T}>) \prec mtype(m, C<\overline{S}>)$.

Examples.

$A<B> \prec A<dyn> \Rightarrow mtype(m, A<B>) \prec mtype(m, A<dyn>)$.

Proof.

Induction based on where m is defined (n classes upwards in hierarchy from C).

Base case (m defined in C):

Follows from preservation of $\prec_{=}$ by type substitution ($\overline{T} \prec_{=} \overline{S} \Rightarrow [\overline{T}/\overline{X}]U \prec_{=} [\overline{S}/\overline{X}]U$) (by lemma 27).

Induction step (m defined in superclass):

Use a similar substitution argument.

Lemma 34. If $C<\overline{T}> \prec_{co} D<\overline{S}>$ and $mtype(m, D<\overline{S}>) = \overline{U} \to U_0$, then $mtype(m, C<\overline{T}>) = \overline{V} \to V_0$ such that $\#(\overline{U}) = \#(\overline{V})$.

Proof.

Easy. From definition of $\prec_{co}$ and $map_\uparrow$ we see that C is a subclass of D. From definition of mtype and validOverride we get the rest.

Corollary.

If $C<\overline{T}> \prec_{co} D<\overline{S}>$, $C<\overline{T}>$ supports all methods that $D<\overline{S}>$ supports and with the same number of arguments.

Lemma 35. If $C<\overline{T}> <: D<\overline{S}>$ then $mtype(m, D<\overline{S}>) \prec_{=} mtype(m, C<\overline{T}>)$,

Proof.

This lemma follows from properties of validOverride, subtyping and method typing, and transitivity of $\prec_{=}$ (lemma 18), and preservation of $\prec_{=}$ by substitution (lemma 27), plus a few minor lemmas.

Proof by induction on the depth of inheritance hierarchy between C and D.

Base case C = D:

Trivial.

Induction step:

Assume class $C<\overline{X}>$ extends $E<\overline{V}>$ { ... }.

We first show that $mtype(m, map_\uparrow(C<\overline{T}>, E)) = mtype(m, [\overline{T}/\overline{X}]E<\overline{V}>) \prec_{=} mtype(m, C<\overline{T}>)$.

If m is present in C:

Now $mtype(m, C<\overline{T}>) = [\overline{T}/\overline{X}](\overline{U} \to U_0)$, where $\overline{U} \to U_0$ is the signature of m in C.

Also $mtype(m, C<\overline{X}>) = \overline{U} \to U_0$.

Also $validOverride(C<\overline{X}>, m, \overline{U} \to U_0) \Rightarrow$ $mtype(m, [\overline{X}/\overline{X}]E<\overline{V}>) = mtype(m, E<\overline{V}>) = \overline{W} \to W_0$ and $\overline{W} \prec_{=} \overline{U}$ and $W_0 \prec_{=} U_0$ (from method typing).

We note that $mtype(m, C<\overline{T}>)$ can be written as $mtype(m, [\overline{T}/\overline{X}]C<\overline{X}>)$.

We can use lemma 27 and lemma 28 to show that if $mtype(m, [\overline{T}/\overline{X}]E<\overline{V}>) = \overline{R} \to R_0$ then $(\overline{R} \to R_0) \prec_{=} [\overline{T}/\overline{X}](\overline{U} \to U_0) = mtype(m, C<\overline{T}>)$.

$mtype(m, [\overline{T}/\overline{X}]E<\overline{V}>) = mtype(m, map_\uparrow(C<\overline{T}>, E))$ (by definition of $map_\uparrow$), i.e. we have shown that $mtype(m, map_\uparrow(C<\overline{T}>, E)) \prec_{=} mtype(m, C<\overline{T}>)$.

else (m is not present in C):

Now $mtype(m, C<\overline{T}>) = mtype(m, map_\uparrow(C<\overline{T}>, E))$, from which the desired conclusion immediately follows.

By lemma 2 and lemma 3, $map_\uparrow(C<\overline{T}>, E) \prec_{=} D<\overline{S}>$.

By induction hypothesis, $mtype(m, D<\overline{S}>) \prec_{=} mtype(m, map_\uparrow(C<\overline{T}>, E))$.

Now, by transitivity of $\prec_{=}$ (lemma 18), $mtype(m, D<\overline{S}>) \prec_{=} mtype(m, C<\overline{T}>)$.

Lemma 36. If $C<\overline{T}> \prec_{=} C<\overline{S}>$, then $mtype(m, C<\overline{T}>) \prec_{=} mtype(m, C<\overline{S}>)$.

Proof.

The proof is similar for each argument type and the return type.

The lemma follows from preservation of $\prec_{=}$ by substitution and the definition of mtype (lemma 27).

6 Properties of fields

Lemma 37. If $C<\overline{T}> \prec_{co} D<\overline{S}>$ and $fields(D<\overline{S}>) = \overline{U}\ \overline{f}$ and $fields(C<\overline{T}>) = \overline{V}\ \overline{g}$, then $\overline{f}$ is a subset of $\overline{g}$.

Proof.

Easy, from definition of fields (using induction). Assume acyclic inheritance hierarchy.

Lemma 38. If $C<\overline{T}> \prec_{=} C<\overline{S}>$ and $fields(C<\overline{T}>) = \overline{U}\ \overline{f}$ and $fields(C<\overline{S}>) = \overline{V}\ \overline{f}$, then $\overline{U} \prec_{=} \overline{V}$.

Proof.

The proof is similar for each U_i/V_i.

The lemma follows from preservation of $\prec_{=}$ by substitution and the definition of fields (lemma 27).

Lemma 39. If $C<\overline{T}> <: D<\overline{S}>$ and $fields(C<\overline{T}>) = \overline{U}\ \overline{f}$ and $fields(D<\overline{S}>) = \overline{V}\ \overline{g}$, then for i in 1, ..., $\#(\overline{U})$, $U_i = V_i$.

Proof.

Proof by induction on inheritance hierarchy depth between C and D.

Base case (C = D):

Trivial.

Induction step:

Assume class $C<\overline{X}>$ extends $E<\overline{V}>$ { ... }.

By lemma 3, $D<\overline{S}> = map_{\uparrow}(C<\overline{T}>, D)$. Also by definition of $map_{\uparrow}$, $E<\overline{U}> = map_{\uparrow}(C<\overline{T}>, E)$ and $D<\overline{S}> = map_{\uparrow}(E<\overline{U}>, D)$.

By definition of fields, $field(C<\overline{T}>)$ has the fields of E as a prefix. By definition of $map_{\uparrow}$ and fields, the shared fields have types equivalent to $fields(E<\overline{U}>)$ $\Rightarrow$ the lemma applies to $C<\overline{T}>$ and $E<\overline{U}>$.

Now apply induction hypothesis to $E<\overline{U}>$ and $D<\overline{S}>$ and combine with the above to finish proof. ■

7 Properties of join

Lemma 40. If $C<\overline{T}> \prec_{co} D<\overline{S}>$ and $join(C<\overline{T}>, D<\overline{S}>) = D<\overline{U}>$, then $D<\overline{U}> \prec_{=} D<\overline{S}>$.

Proof.

By induction on inheritance hierarchy depth between C and D.

Base case $C<\overline{T}>, C<\overline{S}>$:

By lemma 41 (below).

Case $C<\overline{T}>, D<\overline{S}>$, C inherits via n + 1 levels from D, fine for 0..n:

Assume E is the direct superclass of C. Let $E<\overline{U}> = map_{\uparrow}(C<\overline{T}>, E)$.

Now $E<\overline{U}> \prec_{co} D<\overline{S}>$ (by lemma 20) and $join(C<\overline{T}>, D<\overline{S}>) = join(E<\overline{U}>, D<\overline{S}>)$ (by lemma 42).

We can apply induction hypothesis on $join(E<\overline{U}>, D<\overline{S}>)$ $\Rightarrow join(E<\overline{U}>, D<\overline{S}>) \prec_{=} D<\overline{S}>$.

From this the desired conclusion immediately follows: $join(C<\overline{T}>, D<\overline{S}>) \prec_{=} D<\overline{S}>$. ■

Lemma 41. If $C<\overline{T}> \prec_{co} C<\overline{S}>$ and $join(C<\overline{T}>, C<\overline{S}>) = C<\overline{U}>$, then $C<\overline{U}> \prec_{=} C<\overline{S}>$.

Proof.

By induction on maximum depth of $C<\overline{T}>$ and $C<\overline{S}>$.

Base case (C, C):

Trivial.

Induction step $(C<\overline{T}>, C<\overline{S}>)$:

$join(C<\overline{T}>, C<\overline{S}>) = C<join(\overline{T}, \overline{S})>$ (by definition of join).

$\Rightarrow \overline{T} \prec_{=} \overline{S}$
$\Rightarrow \overline{T} \prec_{co} \overline{S}$
$\Rightarrow join(\overline{T}, \overline{S}) = \overline{U}, \overline{U} \prec_{=} \overline{S}$
$\Rightarrow C<\overline{U}> \prec_{=} C<\overline{S}>$ (by definition of $\prec_{=}$)
$\Rightarrow join(C<\overline{T}>, C<\overline{S}>) \prec_{=} C<\overline{S}>$. ■

Lemma 42. If $C<\overline{T}> \prec_{co} D<\overline{S}>$ and $E<\overline{U}> = map_{\uparrow}(C<\overline{T}>, E)$, $E<\overline{U}> \prec_{co} D<\overline{S}>$, then $join(C<\overline{T}>, D<\overline{S}>) = join(E<\overline{U}>, D<\overline{S}>)$.

Proof.

Let $D<\overline{V}> = map_{\uparrow}(C<\overline{T}>, D)$.

$join(C<\overline{T}>, D<\overline{S}>) = D<join(\overline{T}, \overline{V})>$.

$D<\overline{W}> = map_{\uparrow}(map_{\uparrow}(C<\overline{T}>, E), D) = map_{\uparrow}(C<\overline{T}>, D)$.

Now $join(E<\overline{U}>, D<\overline{S}>) = D<join(\overline{W}, \overline{S})> = join(C<\overline{T}>, D<\overline{S}>)$. ■

Lemma 43. If $T \prec S$, then $T <: join(T, S)$.

Examples.

Assume class $C<X>$ extends $D<X>$ { ... }.

$C<A> \prec D<A> \Rightarrow join(C<A>, D<A>) = D<A>$.
$C<A> \prec D<dyn> \Rightarrow join(C<A>, D<dyn>) = D<A>$.

Proof.

Only focus on the last rule in join definition (others are trivial).

So we need to show that if $C<\overline{T}> \prec D<\overline{S}>$ then $C<\overline{T}> <: join(C<\overline{T}>, D<\overline{S}>)$.

Let $D<\overline{U}> = map_{\uparrow}(C<\overline{T}>, D)$.

By lemma 21: $C<\overline{T}> \prec D<\overline{S}> \Rightarrow map_{\uparrow}(C<\overline{T}>, D) \prec D<\overline{S}>$, which is equivalent to $D<\overline{U}> \prec D<\overline{S}>$.

Now $\overline{U} \prec_{=} \overline{S}$ (by lemma 14 and definition of $\prec_{=}$).

$join(C<\overline{T}>, D<\overline{S}>) = D<join(\overline{U}, \overline{S})>$ (by definition of join).

Now, by lemma 45, $join(\overline{U}, \overline{S}) = \overline{U}$
$\Rightarrow join(C<\overline{T}>, D<\overline{S}>) = D<\overline{U}> = map_{\uparrow}(C<\overline{T}>, D)$
$\Rightarrow C<\overline{T}> <: join(C<\overline{T}>, D<\overline{S}>)$ (by lemma 2).

Lemma 44. If $C<\overline{T}> \prec D<\overline{S}>$ then $C<\overline{T}> \prec join(C<\overline{T}>, D<\overline{S}>)$.

Proof.

By lemma 43, $C<\overline{T}> <: join(C<\overline{T}>, D<\overline{S}>)$. Now by lemma 12, $C<\overline{T}> \prec join(C<\overline{T}>, D<\overline{S}>)$. ■

Lemma 45. If $T \prec_{=} S$, then $join(T, S) = T$.

Proof.

We have the following cases.

Case $T \prec_{=} dyn$:

Trivial.

Case T $\prec_=$ T:

Trivial.

Case C<$\overline{T}$> $\prec_=$ C<$\overline{S}$>:

By lemma 12, C<$\overline{T}$> $\prec$ C<$\overline{S}$>. Now by lemma 43,
C<$\overline{T}$> <: join(C<$\overline{T}$>, C<$\overline{S}$>). By definition of join and map$_\uparrow$,
join(C<$\overline{T}$>, C<$\overline{S}$>) = C<$\overline{U}$>. Now by definition of <:,
C<$\overline{T}$> must be equal to C<$\overline{U}$>. ■

8 Properties of `meet`

Lemma 46. If depth(T) $\le$ n and depth(S) $\le$ n, then
depth(meet(T, S)) $\le$ n (if meet exists).

Proof.

Proof by induction on maximum depth of T and S.

Base cases (depth $\Leftarrow$ 1):

case C, D:

It is easy to see that meet(C, D) must be either C or D, from which the result immediately follows.

Otherwise:

Trivial.

Induction step:

Case C<$\overline{T}$>, dyn:

Trivial.

Case dyn, C<$\overline{T}$>:

Trivial.

Case C<$\overline{T}$>, D<$\overline{T}$>:

Without loss of generality, assume C is a subclass of D.

By definition of meet, D<$\overline{U}$> = map$_\uparrow$(C<$\overline{T}$>, D), and
by lemma 5, depth(D<$\overline{U}$>) $\le$ depth(C<$\overline{T}$>).

Use induction hypothesis and the above property on meet($\overline{U}$, $\overline{S}$)
$\Rightarrow$ for all i, depth(meet(U_i, S_i)) $\le$ depth(T_i) (and similar for S_i). (1)

Now by lemma 6, depth(map$_\downarrow$(D<meet($\overline{U}$, $\overline{S}$)>, C, $\overline{T}$) $\le$
max(depth(D<meet($\overline{U}$, $\overline{S}$)>, C<$\overline{T}$>) $\le$
max(depth(C<$\overline{T}$>), depth(D<$\overline{S}$>)) (use equation 1). ■

Lemma 47. If T $\prec_{co}$ S then meet(T, S) = T.

Notes.

T $\prec_{co}$ S also implies that meet(T, S) exists.

Proof.

Proof by induction on maximum depth of T and S.

Base cases (depth $\Leftarrow$ 1):

Case dyn $\prec_{co}$ dyn:
Trivial.
Case X $\prec_{co}$ X:
Trivial.
Case X $\prec_{co}$ dyn:
Trivial.
Case C $\prec_{co}$ dyn:
Trivial.
Case C $\prec_{co}$ D:
Easy (due to depth assumption).

Case C<$\overline{T}$>, D<$\overline{S}$>:

Assume, without loss of generality, that C<$\overline{T}$> $\prec_{co}$ D<$\overline{S}$> $\Rightarrow$ C is not a superclass of D.

By definition of meet:

Let map$_\uparrow$(C<$\overline{T}$>, D) = D<$\overline{U}$> and
D<$\overline{V}$> = D<meet($\overline{U}$, $\overline{S}$)>.
Now meet(C<$\overline{T}$>, D<$\overline{S}$>) = map$_\downarrow$(D<$\overline{V}$>, C, $\overline{T}$).

By definition of $\prec_{co}$, $\overline{U}$ $\prec_{co}$ $\overline{S}$. By lemma 5,
depth(D<$\overline{U}$>) $\le$ depth(C<$\overline{T}$>); by lemma 4,
depth($\overline{U}$) < depth(D<$\overline{U}$>) and depth($\overline{T}$) < depth(C<$\overline{T}$>).
Now by induction hypothesis, meet($\overline{U}$, $\overline{S}$) = $\overline{U}$ $\Rightarrow$ D<$\overline{V}$> = D<$\overline{U}$>.

Now meet(C<$\overline{T}$>, D<$\overline{S}$>) = map$_\downarrow$(map$_\uparrow$(C<$\overline{T}$>, D), C, $\overline{T}$)
= C<$\overline{T}$> (by lemma 10). ■

Lemma 48. meet(T, S) $\prec_{co}$ T, if meet exists.

Notes.

Assume class C<X> extends D<X, X> { ... }.

Now meet(C<dyn>, D<A, dyn>) is undefined, but neither D<a, dyn> $\prec_{co}$ C<dyn> nor C<dyn> $\prec_{co}$ D<a, dyn>, so this is fine.

Proof.

Proof by induction maximum on maximum depth of T and S.

Each case is tuple (T, S), where T and S are as in lemma definition.

Base cases:

Case dyn, T:

Trivial.

Case T, dyn:

Trivial.

Case T, T:

Trivial.

Case C, D when C $\neq$ D:

Easy (due to depth assumption).

Induction step:

Case $C<\overline{T}>$, dyn:

Trivial.

Case dyn, $C<\overline{T}>$:

Trivial.

Case $C<\overline{T}>, D<\overline{S}>$:

By lemma 49, either C is a subclass of D or D is a subclass of C. Assume, without loss of generality, that C is a subclass of D (switch $C<\overline{T}>$ and $D<\overline{S}>$ if needed).

By definition of meet:

Let $\mathrm{map}_{\uparrow}(C<\overline{T}>, D) = D<\overline{U}>$ and
$D<\overline{V}> = D<\mathrm{meet}(\overline{U}, \overline{S})>$.
Now $\mathrm{meet}(C<\overline{T}>, D<\overline{S}>) = \mathrm{map}_{\downarrow}(D<\overline{V}>, C, \overline{T})$.

We also need lemma 46.

By lemma 5, $\mathrm{depth}(D<\overline{U}>) \leq \mathrm{depth}\ C<\overline{T}>$.

We can now use the induction hypothesis on $\overline{U}$ and $\overline{S} \Rightarrow$
$\overline{V} \prec_{co} \overline{U}$ and $\overline{V} \prec_{co} \overline{S}$
$\Rightarrow D<\overline{V}> \prec_{co} D<\overline{U}>$ and $D<\overline{V}> \prec_{co} D<\overline{S}>$
(by definition of $\prec_{co}$).

Now $D<\overline{V}> \prec_{co} \mathrm{map}_{\uparrow}(C<\overline{T}>, D) = D<\overline{U}>$
$\Rightarrow \mathrm{map}_{\downarrow}(D<\overline{V}>, C, \overline{T}) \prec_{co} C<\overline{T}>$ (by lemma 24; this exists, since we assume that $\mathrm{meet}(C<\overline{T}>, D<\overline{S}>$ exists)
$\Rightarrow \mathrm{meet}(C<\overline{T}>, D<\overline{S}>) \prec_{co} C<\overline{T}>$.

Also, since $D<\overline{V}> \prec_{co} D<\overline{S}>$, $\mathrm{map}_{\downarrow}(D<\overline{V}>, C, \overline{T}) \prec_{co} D<\overline{S}>$ (by lemma 23 and transitivity of $\prec_{co}$). ■

<u>Lemma 49</u>. $\mathrm{meet}(C<\overline{T}>, D<\overline{S}>)$ exists only if C is a subclass of D or D is a subclass of C.

<u>Proof</u>.

By definition of meet, either $\mathrm{map}_{\uparrow}(C<\overline{T}>, D)$ or $\mathrm{map}_{\uparrow}(D<\overline{S}>, C)$ must exist $\Rightarrow$ either C must be a subclass of D or D must be a subclass of C. ■

<u>Definition 8</u>.

$!C<\overline{T}> = \overline{T}$.

Examples.

$!\mathrm{meet}(C<\overline{T}>, C<\overline{S}>) = \mathrm{meet}(\overline{T}, \overline{S})$.

<u>Lemma 50</u>. $\mathrm{meet}(S, \mathrm{meet}(U, T)) = \mathrm{meet}(S, T)$, if $S \prec_{co} U$ and the LHS exists.

<u>Proof</u>.

Proof by induction on the maximum depth of S, U and T.

Omit cases with type variables (they are all trivial).

Base cases:

Case dyn, dyn, dyn:
Trivial.

Case dyn, dyn, C:
Trivial.

Case C, dyn, dyn:
Trivial.

Case C, dyn, D:
Trivial.

Case C, D, dyn:
Easy; use lemma 47.

Case C, D, E:
It is easy to see that meet(D, E) is either D or E (we can assume that it exists). In the former case, the result is D (it is fairly easy to see that also meeet(C, E) must equal D); in the latter case the result is meet(D, E).

Induction step:

Case $C<\overline{T}>$, dyn, dyn:
Trivial.

Case $C<\overline{T}>$, dyn, $E<\overline{U}>$:
Trivial.

Case $C<\overline{T}>, D<\overline{S}>$, dyn:
Easy; use lemma 47.

Case $C<\overline{T}>, D<\overline{S}>, E<\overline{U}>$:

$\mathrm{meet}(C<\overline{T}>, \mathrm{meet}(D<\overline{S}>, E<\overline{U}>))$ can be written like this using the alternative notation introduced earlier (use definition of meet):

$$\begin{aligned}
&\mathrm{meet}(C<\overline{T}>, \mathrm{meet}(D<\overline{S}>, E<\overline{U}>) = \\
&\quad C<\mathrm{meet}(\overline{T}{:}C/D,\ !\mathrm{meet}(D<\overline{S}>, E<\overline{U}>)){:}D/C<\overline{T}>> \\
&\quad = C<\mathrm{meet}(\overline{T}{:}C/D, \\
&\qquad\qquad \mathrm{meet}(\overline{S}{:}D/E, \overline{U}){:}E/D<\overline{S}>){:}D/C<\overline{T}>>
\end{aligned}$$

Now by lemma 51; also use lemma 22 (assuming the original exists):

$$\begin{aligned}
&= C<\mathrm{meet}(\overline{T}{:}C/E, \\
&\qquad \mathrm{meet}(\overline{S}{:}D/E, \overline{U})){:}E/C<\overline{T}>>
\end{aligned}$$

Use induction hypothesis to simplify away the second meet:

$$\begin{aligned}
&= C<\mathrm{meet}(\overline{T}{:}C/E, \overline{U}){:}E/C<\overline{T}>> \\
&= \mathrm{meet}(C<\overline{T}>, E<\overline{U}>) \text{ (by definition of meet),}
\end{aligned}$$

which is the desired result. ■

<u>Lemma 51</u>. If $\mathrm{meet}(\overline{T}, \overline{U}{:}D/C<\overline{S}>) = \mathrm{meet}(\overline{T}{:}C/D, \overline{U}){:}D/C<\overline{T}>$, if the LHS is defined, C is a subclass of D and $\overline{T} \prec_{co} \overline{S}$.

<u>Proof</u>.

The LHS is equivalent to $\mathrm{meet}(\overline{T}, \overline{U}{:}D/C<\overline{T}>)$ by lemma 52.

Then note that the RHS can be written like this, since $U{:}D/C<\overline{T}>$ exists, by lemma 8:

$meet(\overline{T}:C/D,\ \overline{U}:D/C<\overline{T}>:C/D):D/C<\overline{T}>$

Now we can use lemma 53 to show that the LHS and RHS are equal.

Lemma 52. $meet(\overline{T},\ \overline{U}:D/C<\overline{S}>) = meet(\overline{T},\ \overline{U}:D/C<\overline{T}>)$, if C is a proper subclass of D and $\overline{T} \prec_{co} \overline{S}$ and the LHS exists.

Proof.

Assume class $C<\overline{X}>$ extends ...

We focus on $\overline{V} = \overline{U}:D/C<\overline{S}>$ and $\overline{W} = \overline{U}:D/C<\overline{T}>$.

Some $V_i = W_i$, if X_i in C is mapped all the way up to D (in this case, $\overline{S}$ and $\overline{T}$ have no effect). Now trivially $meet(T_i,\ V_i) = meet(T_i,\ W_i)$.

If $V_i \neq W_i$, then some $map_{\downarrow}$ step (at class F) has introduced a component from $map_{\uparrow}(C<\overline{S}>, F)$ or $map_{\uparrow}(C<\overline{T}>, F)$. But by lemma 22, and since $map_{\downarrow}$ performs the same transformation for type arguments independent of the value of the type argument, we can see that $W_i \prec_{co} V_i \prec_{co} T_i$.
Now $meet(T_i,\ W_i) = T_i = meet(T_i,\ V_i)$.

By combining the above, $meet(\overline{T},\ \overline{V}) = meet(\overline{T},\ \overline{W})$. ■

Lemma 53. $meet(\overline{T}:C/D,\ \overline{S}:D/C<\overline{T}>:C/D):D/C<\overline{T}> = meet(\overline{T},\ \overline{S}:D/C<\overline{T}>)$, assuming the LHS is defined and C is a subclass of D.

Proof.

Let LHS = $\overline{U}$ and RHS = $\overline{V}$.

Now some U_i are derived from $\overline{T}$, while some are derived from meet(...), which is easy to see by observing the definition of $map_{\downarrow}$.

We deal with these two classes of U_i separately.

If U_i is derived from $\overline{T}$:

If U_i is derived from $\overline{T}$, $U_i = T_i$; by similar reasoning also $V_i = T_i$ ($V_i = meet(T_i,\ T_i)$).

If U_i is derived from meet(...):

In this case it is easy see that the substitution does not affect the value of the meet (substitution preserves the value of meet).

9 Safe coercion lemmas

Definition 9. (safe coercions and casts)

A coercion or cast is safe if it cannot be a target of blame (i.e. it preserves the blame label of the target value) and if it always succeeds.

Lemma 54. Coercion $<T \Leftarrow S><S>new\ C<\overline{U} \Leftarrow \overline{V}>(...)$ is safe if $C<\overline{U}> \prec T$.

Proof.

If the coercion is successful, $meet(C<\overline{U}>,\ T) = C<\overline{W}>$.

Now since $C<\overline{U}> \prec T$, $meet(C<\overline{U}>,\ T) = C<\overline{U}>$ (by lemma 47 and lemma 13)
$\Rightarrow$ the coercion is successful and the meet exists.

Since $C<\overline{U}> \prec T$, coercion clearly preserves the blame label (E-Coerce).

Lemma 55. Cast $(T)<S>new\ C<\overline{U} \Leftarrow \overline{V}>(...)$ is safe if $C<\overline{U}> \prec T$.

Proof.

Similar to lemma 54 above.

Definition 10. Coercion $<T \Leftarrow S><S>new\ C<\overline{U} \Leftarrow \overline{V})(...)$ is trivial if $S \prec T$.

Examples.

$<C \Leftarrow C>$ trivial
$<C \Leftarrow D>$ trivial if D subclass of C
$<dyn \Leftarrow C>$ trivial
$<C<D> \Leftarrow C<D>>$ trivial
$<C<dyn> \Leftarrow C<D>>$ trivial

$<C<D> \Leftarrow C<E>>$ not trivial if $D \neq E$
$<C<D> \Leftarrow C<dyn>>$ not trivial

Notes.

We prove later that trivial coercions are safe (lemma 61).

10 Transformation lemmas

Lemma 56. (transformation preserves typing) If $\Delta,\ \Gamma \vdash e : T$ and $\Delta,\ \Gamma \vdash e \rightsquigarrow e' : T$, then $\Delta,\ \Gamma \vdash e' : T$ (using alternative typing rules).

Proof.

By induction on transformation relation.

Bases cases are trivial.

Induction step:

Case Tr-Invk:

We coerce the arguments based on method signaature (as expected by TT-GInvk); otherwise trivial.

If argument coercion is unsafe:

We can blame the call, but that is valid; arguments are not compatible. The blame happens either immediately or later, but in any case, the blame label is valid.

If argument coercion is safe:

We will not blame this coercion. The coercion succeeds, and the label will not be kept with the instance.

Case Tr-DyInvk:

We coerce the argument to dyn (as expected by TT-DyGInvk); otherwise trivial.

Case Tr-Create:

We coerce the arguments to correct types. The conditions $\overline{T} \prec_{co} \overline{S}$ and $C<\overline{T}> \prec_{co} U$ of TT-Creat are also satisfied; the rest is trivial.

Case Tr-Cast:

Trivial.

Case Tr-Field:

Trivial.

Case Tr-Var:

Trivial.

Additional cases:

Method transform:

Trivial; preserves method signatures.

Class transform:

Trivial; only affects methods, everything else is preserved.

Definition 11. (valid blame label)

The label ℓ is valid target for blame if it refers to either:

* an invocation which used unsafe coercion for arguments; blame invoke
* a new expression which used unsafe coercion for arguments; blame new
* an invocation or field access where the receiver type is not precisely typed (e.g. dyn or A<dyn>); blame invocation or field access
* a method body where return used an unsafe coercion; blame method
* a method override that is unsafe; blame the override
* a downcast.

And, in particular, the empty label is not a valid blame target.

Lemma 57. Whenever transformation produces a non-trivial coercion, the coercion blame label is valid (by definition 11).

Proof.

By induction on transformation relation.

Base cases are trivial; we only go through the induction step.

Case Tr-Invk:

We blame the call expression if an argument needs an unsafe coercion based on the static method signature. We also keep track of the blame label in the guarded invoke, and we may blame the original invoke during evaluation if receiver type is inprecise. We prove this case later.

Case Tr-DyInvk:

Since the argument target type is dyn, the coercion is always trivial; we can use an empty label.

The guarded invoke has the label of the original invoke. This is blamed if the argument types do not match the actual called method signature, which is valid (as receiver has type dyn).

Case Tr-Creat:

We blame the new expression if an argument needs an unsafe coercion based on static constructor signature.

Case Tr-Field:

We keep track of the field expression label. We may blame it if the receiver type is imprecise. We deal with this case later.

Method transform:

We blame the method if the return value needs unsafe coercion, which is valid.

Otherwise:

Trivial.

11 Type preservation lemma

Lemma 58. Evaluation preserves types.

More precise formulation:

Assume Δ, $\Gamma \vdash$ e : T and e $\longrightarrow$ e'; now Δ, $\Gamma \vdash$ e' : T.

Proof.

By induction on evaluation relation.

Base case:

Transformation preserves typing by lemma 56.

Induction step:

Case E-Coerce-Fail:

Terminates the program, nothing to do.

Case E-GInvk:

Easy. We add all the needed coercions.

Case E-Coerce:

Initially runtime type <U>new C<$\overline{T} \Leftarrow \overline{S}$>; arbitrary type. Target type V (static type).

New runtime type <V>new C<$\overline{W} \Leftarrow \overline{S}$>, where meet(C<$\overline{T}$>, V) = C<$\overline{W}$>.

This preserves typing of value ($\overline{S}$ remains unchanged) and the result type is V, as expected.

By TT-Creat, we also need to preserve $\overline{T} \prec_{co} \overline{S}$:

<V $\Leftarrow$ U><U>new C<$\overline{T} \Leftarrow \overline{S}$>(...)
$\longrightarrow$ <V>new C<$\overline{W} \Leftarrow \overline{S}$>(...) where meet(C<$\overline{T}$>, V) = C<$\overline{W}$>.

$\overline{T} \prec_{co} \overline{S}$ (by induction hypothesis) $\Rightarrow$ C<$\overline{T}$> $\prec_{co}$ C<$\overline{S}$>.

C<$\overline{W}$> = meet(C<$\overline{T}$>, V) $\prec_{co}$ C<$\overline{T}$> (by lemma 48).

Now we have C<$\overline{W}$> $\prec_{co}$ C<$\overline{T}$> $\prec_{co}$ C<$\overline{S}$>

$\Rightarrow$ C<$\overline{W}$> $\prec_{co}$ C<$\overline{S}$> (by lemma 16)

$\Rightarrow$ $\overline{W} \prec_{co} \overline{S}$, as expected.

Similarly, we need to preserve C<$\overline{T}$> $\prec_{co}$ U:

<V ⇐ U><U>new C<$\overline{\mathtt{T}}$ ⇐ $\overline{\mathtt{S}}$>(...)
⟶ <V>new C<$\overline{\mathtt{W}}$ ⇐ $\overline{\mathtt{S}}$>(...) where meet(C<$\overline{\mathtt{T}}$>, V) = C<$\overline{\mathtt{W}}$>.

Now C<$\overline{\mathtt{W}}$> $\prec_{\mathtt{co}}$ V (by lemma 48), as expected.

Case E-FieldAcc:

Case receiver type dyn:

Trivial.

Case static receiver type <D<$\overline{\mathtt{U}}$>>C<$\overline{\mathtt{T}}$ ⇐ $\overline{\mathtt{S}}$>:

Easy.

Case E-Invk:

First, we need to show that substitution preserves typing (by lemma 59).

Then, the result should have the correct type.

It is easy to see that method body has same return type W as that from mtype(m, C<$\overline{\mathtt{X}}$>) (by TT-Method and definition of mtype).

Now the result has type [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]W from substitution lemma, which coincides with that of TT-Invk.

By TT-Creat, we also need to preserve $\overline{\mathtt{T}} \prec_{\mathtt{co}} \overline{\mathtt{S}}$:

We have to show that substitution preserves the property.

Substitution only applies to method body; in it all cases of <U>new C<...> are trivially of form <U>new C<U ⇐ U>(...) (due to Tr-Creat), so these are trivialy preserved.

Additionally, we substitute this with receiver; since receiver has the property, this is fine. Substitution of arguments is similarly trivial.

We also need to preserve C<$\overline{\mathtt{T}}$> $\prec_{\mathtt{co}}$ U; for this we use similar reasoning as above.

Case E-Cast:

This is similar to E-Coerce above.

By TT-Creat, we also need to preserve $\overline{\mathtt{T}} \prec_{\mathtt{co}} \overline{\mathtt{S}}$:

Similar to E-Coerce above.

Similarly, we need to preserve C<$\overline{\mathtt{T}}$> $\prec_{\mathtt{co}}$ U:

Similar to E-Coerce above.

Other case:

Trivial.

Lemma 59. If mbody(m, C<$\overline{\mathtt{X}}$>) = $\overline{\mathtt{x}}$. e, cargs(C) = $\overline{\mathtt{X}}$ and <C<$\overline{\mathtt{S}}$>>new C<$\overline{\mathtt{T}}$ ⇐ $\overline{\mathtt{S}}$>$_\ell$($\overline{\mathtt{v}}$).m($\overline{\mathtt{u}}$) is well-typed, then [$\overline{\mathtt{u}}$/$\overline{\mathtt{x}}$, <C<$\overline{\mathtt{S}}$>>new C<$\overline{\mathtt{T}}$ ⇐ $\overline{\mathtt{S}}$>$_\ell$($\overline{\mathtt{v}}$)/this, $\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]e is well-typed.

Proof.

By induction on depth of e; assume well-typed and if original type is T, new type is [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]T.

Base cases:

Case x:

Due to substitution, new value is $\mathtt{u_i}$ and this is well-typed (due to assumptions). Original type was from mtype(m, C<$\overline{\mathtt{X}}$>); new type is from mtype(m, C<$\overline{\mathtt{S}}$>). Substituting [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$] results in the new type, as required.

Case this:

Original type C<$\overline{\mathtt{X}}$>; after substitution C<[$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]$\overline{\mathtt{X}}$> = C<$\overline{\mathtt{S}}$>.

The new type is C<$\overline{\mathtt{S}}$> (fine); the replacement is also well-typed (due to assumptions).

Case <U>new C<$\overline{\mathtt{V}}$ ⇐ $\overline{\mathtt{W}}$>():

Original type U; after substitution [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]U, which is fine.

Induction step:

Case <U ⇐ V>e:

Type of e is S, new type is [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]V, which is fine. The type of coercion expression is [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]U, as expected.

Case <U>new C<$\overline{\mathtt{V}}$ ⇐ $\overline{\mathtt{W}}$>($\overline{\mathtt{e}}$):

Type of $\overline{\mathtt{E}}$ = $\overline{\mathtt{R}}$ (equals to type of fields(C<$\overline{\mathtt{W}}$>)); after substitution [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]$\overline{\mathtt{R}}$ (by induction hypothesis).

$\overline{\mathtt{E}}$ equals type of fields(C<$\overline{\mathtt{W}}$>).

Now after substitution, $\overline{\mathtt{E}}$ should be equal to types of fields([$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]C<$\overline{\mathtt{W}}$>) = types of [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]fields(C<$\overline{\mathtt{W}}$>), as expected (by lemma 30).

Original type of the new expression is U; the type after substitution is [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]U, as expected.

Case <e.m($\overline{\mathtt{e}}$)>$_\ell$:

Type of e = U, type of $\overline{\mathtt{e}}$ = $\overline{\mathtt{V}}$. The type of method call is the return type via mtype, W. After substitution these are [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]U, [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]$\overline{\mathtt{V}}$ and [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]W, respectively.

By lemma 28, mtype(m, [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]U) = [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]mtype(m, U); then everything else follows.

Case e.f:

Type of e.f is based on type of e and fields(...). Substitution does not affect the existence of f. Subsituting [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$] for type of e.f is equivalent to substituting for receiver, and then looking up fields. By lemma 30, fields([$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]U) = [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]fields(U), as required.

Case (T)e:

Original type is T; type after substitution if [$\overline{\mathtt{S}}$/$\overline{\mathtt{X}}$]T, as required.

Case e.m($\overline{\mathtt{e}}$):

This form is never in the result of mbody(...).

12 Progress lemma

Lemma 60. Evaluation progresses for well-typed programs or causes blame, unless there is a failed downcast.

More precise formulation.

Assume Δ, $\Gamma \vdash$ e : T and e $\neq$ blame ℓ. Now e $\longrightarrow$ e' unless there is a failed downcast.

Proof.

By induction on evaluation relation.

Assume type preservation (lemma 58).

Case E-Coerce, E-Coerce-Fail:

It is easy to see that either there is proper progress (E-Coerce) or there is blame (E-Coerce-Fail).

Case E-GInvk:

Assume receiver is <U>new C<$\overline{T} \Leftarrow \overline{S}$>.

Runtime type C<$\overline{T}$> $\prec_{co}$ U (by TT-Creat and type preservation).

By typing (TT-GInvk, TT-Creat) and type preservation, for static type U, method m exists and has the right argument count.

It is easy to see that C<$\overline{T}$> and C<$\overline{S}$> support the same methods.

For C<$\overline{T}$> $\prec_{co}$ U, C<$\overline{T}$> supports all methods that U supports and has the same number of arguments (by lemma 34).

So by this progress is easy.

Case E-DyGInvk:

We blame the call if the method does not exist, which is fine. Otherwise, we proceed with evaluation.

Case E-Invk:

If the receiver type is not dyn, use similar reasoning as for E-GInvk.

For dyn receiver:

If we reach this, the call is valid, from the premises of E-DyGInvk. We only reach this after E-DyGinvk, due to properties of transformation (Tr-DyInvk).

Case E-FieldAcc:

Assume receiver <U>new C<$\overline{T} \Leftarrow \overline{S}$>.

Actual receiver runtime type C<$\overline{T}$> $\prec_{co}$ U (by TT-Creat).

It is easy so see that C<$\overline{T}$> and C<$\overline{S}$> have the same fields.

By typing (T-Field, TT-Creat) and type preservation, U has the field f.

For C<$\overline{T}$> $\prec_{co}$ U, C<$\overline{T}$> has all fields U has (by lemma 37; U is an instance type).

So by this progress is easy.

Case E-DyFieldAcc:

We blame the field access if the field does not exist.

Case E-DynCast:

Trivial.

Case E-Cast:

If the cast is an upcast:

(D<$\overline{V}$>)<U>new C<$\overline{T} \Leftarrow \overline{S}$>(...) where U <: D<$\overline{V}$> (from definition of upcast, T-CastUp and TT-Create). Also C<$\overline{T}$> $\prec_{co}$ U (by TT-Creat).

From C<$\overline{T}$> $\prec_{co}$ U and U <: D<$\overline{V}$>, also C<$\overline{T}$> $\prec_{co}$ D<$\overline{V}$> (by lemma 12 and lemma 16).

Now meet(C<$\overline{T}$>, D<$\overline{V}$>) = C<$\overline{T}$> (lemma 47)
$\Rightarrow$ Premises are met $\Rightarrow$ There is progress.

For downcasts, there may not be progress (by original assumptions).

13 The blame theorem

Definition 12. (unsafe method override)

Method override m in C is unsafe if return type of mtype(m, C) $\neq$ return type of mtype(m, D), where D is the superclass of C.

Theorem 1. (blame) If e $\longrightarrow$ blame ℓ, then we can validly blame ℓ (according to definition 11).

Proof.

First we use lemma 57. We also use lemma 58.

Proof by induction on evaluation relation.

Additional preserved properties (part of the induction hypothesis):

Property 1. All unsafe coercions generated during coercion have a valid blame label.

Property 2. If the label of a new expression <U>new C<$\overline{T} \Leftarrow \overline{S}$>(...) is empty, C<$\overline{T}$> $\prec$ U.

Property 3. If the blame label of a new expression is non-empty, it has been through an unsafe coercion or an unsafe downcast, and the blame label refers to an unsafe coercion/cast.

Property 4. If label is empty in <U>new C<$\overline{T} \Leftarrow \overline{S}$>(...), then $\overline{S}$ = $\overline{T}$.

Initial state (after transformation):

* Use lemma 57.
* Property 2 is trivially true.

* Property 3 is trivially true (all values have empty blame labels).
* Property 4 is trivially true.

We prove the preservation of property 1 and the main theorem first, and the preservation of properties 2, 3 and 4 separately.

We also use the lemma below. It assumes some of the above properties.

Lemma 61. A trivial coercion $<T \Leftarrow S>e$ is safe (i.e. if $S \prec T$).

Proof.

Assume coercion $<T \Leftarrow S><S>\text{new } C<\overline{U} \Leftarrow \overline{V}>(\ldots)$ (omitting labels).

By TT-Creat and type preservation, $C<\overline{U}> \prec_{co} S$.

$S \prec T$ by assumptions, and by lemma 13 and lemma 16, $C<\overline{U}> \prec_{co} T$.

If the coercion is successful, $\text{meet}(C<\overline{U}>, T) = C<\overline{W}>$ (by E-Coerce).

Now since $C<\overline{U}> \prec_{co} T$, $\text{meet}(C<\overline{U}>, T) = C<\overline{U}>$ (by lemma 47 and lemma 13) $\Rightarrow$ the coercion is successful and the meet exists.

We preserve the value blame label if ℓ is not empty or $C<\overline{U}> \prec T$ (by E-Coerce). If the label is not empty, the coercion is thus safe.

Consider an empty value blame label. By property 2, $C<\overline{U}> \prec S$. By transitivity of $\prec$, $C<\overline{U}> \prec T \Rightarrow$ the empty blame label is preserved and the coercion is safe. ■

Induction step (main properties):

Case E-Coerce:

The result blame label can be from the coercion or from the value.

If from the value ($\ell'' = \ell$):

The value blame label is non-empty or $C<\overline{T}> \prec V$ (or both).

If value blame label is non-empty ($C<\overline{T}> \prec V$ can be anything):

We propagate the the value blame label, which is valid (by induction hypothesis, including property 3).

If value blame label is empty and $C<\overline{T}> \prec V$:

This is a safe coercion (by lemma 54); therefore we can propagate the empty blame label.

If from the coercion ($\ell'' = \ell'$):

The value blame label is empty and not $C<\overline{T}> \prec V$.

This coercion is unsafe due, since $C<\overline{T}>$ not $\prec V$ (by lemma 54). Therefore we can attach the coercion label to the result, and we know that it is not empty (due to induction hypothesis).

Case E-GInvk:

The case is divided into 8 subcases. We go through each of them below.

We use the following definitions below:

$D<\overline{N}> = \text{map}_{\uparrow}(C<\overline{U}>, D)$
$\text{mtype}(D<\overline{N}>) = \overline{O} \to O_0$

Argument cases:

Case $R \Leftarrow W$, blame call:

By TT-Creat and type preservation, $C<\overline{U}> \prec_{co} D<\overline{T}>$.

Now we can use lemma 40: $D<\overline{V}> \prec_{=} D<\overline{T}>$.

Now if $D<\overline{T}>$ has no dyn, $\overline{V} = \overline{T}$ (by lemma 15) $\Rightarrow W = R$ and the coercion is trivial and cannot cause blame (by lemma 61).

If $D<\overline{T}>$ has dyn components, it is a valid blame target, and we can blame the call.

Case $O \Leftarrow R$, empty value:

If value value has an empty blame label:

$C<\overline{U}> \prec D<\overline{T}>$ by property 2.

$\Rightarrow \text{map}_{\uparrow}(C<\overline{U}>, D) \prec D<\overline{T}>$ (by lemma 21),

i.e. $D<\overline{N}> \prec D<\overline{T}>$.

$D<\overline{N}> \prec \text{join}(D<\overline{N}>, D<\overline{T}>)$ by lemma 44.

$D<\overline{N}> <: \text{join}(D<\overline{N}>, D<\overline{T}>) = D<\overline{V}>$ by lemma 43.

$\Rightarrow \overline{N} = \overline{V}$ (by definition of $<:$)

$\Rightarrow$ coercion $O \Leftarrow R$ is safe (by lemma 61).

Therefore the empty blame label is valid.

else (value has non-empty blame label):

We can blame the blame label of the value (by property 3 in the induction hypothesis).

Case $S \Leftarrow O$, empty blame:

The target signature is based on $C<\overline{U}>$; the source signature based on $\text{map}_{\uparrow}(C<\overline{U}>, D)$.

$C<\overline{U}> <: \text{map}_{\uparrow}(C<\overline{U}>, D)$ (by lemma 2)

$\Rightarrow \overline{O} \prec_{=} \overline{S}$ by lemma 35

$\Rightarrow$ coercion $S \Leftarrow O$ is safe (by lemma 61 and property 2).

Therefore the empty blame label is valid.

Case $P \Leftarrow S$, blame value:

If value has empty blame label:

By property 4, $\overline{U} = \overline{Q}$. This implies that $P = S \Rightarrow$ the coercion is trivial, and can never cause blame (by lemma 61). Therefore the empty blame is label is valid.

else (value has non-empty blame label):

We can blame the blame label of the value (by property 3 in the induction hypothesis).

Return value cases:

Case $S_0 \Leftarrow P_0$, blame value:

Similar to case $P \Leftarrow S$ above.

Case $O_0 \Leftarrow S_0$, blame override:

If $O_0 \neq S_0$, the return value type from mtype(m, C<$\overline{U}$>) $\neq$ return value from mtype(m, D<$\overline{N}$>). Due to properties of validOverride, the return value type in the method override must be less specific than the overridden method $\Rightarrow$ it is valid to blame the override.

Case $R_0 \Leftarrow O_0$, blame value:

Similar to case $O \Leftarrow R$ above.

Case $W_0 \Leftarrow R_0$, no blame:

By similar reasoning as for case $R \Leftarrow W$ above, D<$\overline{V}$> $\prec_=$ D<$\overline{T}$>, and by lemma 36, $R_0 \prec_= W_0$.

Also $R_0 \prec W_0$ by property 2, and by lemma 61, the coercion is thus safe. Therefore the empty blame label is valid.

Case E-DyGInvk:

There are four coercions inserted by this rule. We go through them separately.

Case $S \Leftarrow$ dyn (argument, blame call):

The argument type is coerced to the runtime argument type. It is valid to blame the call, since we have an untyped receiver.

Case $W \Leftarrow S$ (argument, blame value):

Similar to the reasoning below ($S_0 \Leftarrow W_0$).

Case $S_0 \Leftarrow W_0$ (return value, blame value):

If value blame label is empty, $S_0 = W_0$ (by property 4) $\Rightarrow$ the coercion is safe, so empty blame label is valid.

If value blame label is not empty, it is a valid blame target (by property 3 in the induction hypothesis).

Case dyn $\Leftarrow S_0$ (return value, empty blame label is valid):

This is a safe coercion, so empty blame is valid.

Case E-Invk:

It is easy to see that this case does not modify blame labels and generate coercions.

Case E-FieldAcc:

There are three coercions generated by this rule. We go through them separately.

Case $W_i \Leftarrow R_i$ (blame value):

Similar to case $S_0 \Leftarrow W_0$ of E-DyGinvk.

Case $P_i \Leftarrow W_i$ (blame value):

First let fields(map$_\uparrow$(C<$\overline{T}$>, D)) = $\overline{O}$ $\overline{f}$.

C<$\overline{T}$> <: map$_\uparrow$(C<$\overline{T}$>, D) by lemma 2.

Now by lemma 39, $\overline{O}$ and $\overline{W}$ are equal (in the common prefix).

Therefore, the coercion is equivalent to $P_i \Leftarrow O_i$.

If the value has an empty blame label:

By property 2, C<$\overline{T}$> $\prec$ D<$\overline{U}$>.

$\Rightarrow$ C<$\overline{T}$> $\prec$ D<$\overline{Q}$> by lemma 44.

By lemma 21, map$_\uparrow$(C<$\overline{T}$>, D) $\prec$ D<$\overline{Q}$>.

$\Rightarrow$ map$_\uparrow$(C<$\overline{T}$>, D) $\prec_=$ D<$\overline{Q}$> (by properties of $\prec_=$ and lemma 14)
$\Rightarrow$ by lemma 38, $W_i \prec_= P_i$
$\Rightarrow$ the coercion is safe, and we can use an empty blame label.

Otherwise:

The blame label is non-empty, and it is a valid blame target in an unsafe coercion (by property 3).

Case $V_i \Leftarrow P_i$ (no blame):

By TT-Creat and type preservation, C<$\overline{T}$> $\prec_{co}$ D<$\overline{U}$>.

Now we can use lemma 40: D<$\overline{Q}$> $\prec_=$ D<$\overline{U}$>.

Now by lemma 38, $P_i \prec_= V_i \Rightarrow$ the coercion is safe, and we can use an empty blame label.

Case E-DyFieldAcc:

The final coercion is safe (target type dyn), so empty label is valid. The first coercion blames the value blame label. If this label is empty, $\overline{T} = \overline{S}$ (by property 4) and the coercion is safe (using lemma 33). If this label is non-empty, we can blame this label if the coercion is unsafe; there has been an unsafe coercion (by property 3). By induction hypothesis, the coercion that produced the blame label had a valid blame label.

Case E-Cast:

Similar to E-Coerce, but use lemma 55 instead of lemma 54.

Case E-DynCast:

Trivial; this is equivalent to a (safe) coercion to dyn.

Case E-DyGInvk-Fail:

We blame method invocation with dyn receiver. This is always valid.

Case E-DyFieldAcc-Fail:

We blame field access with dyn receiver. This is always valid.

Case E-Coerce-Fail:

By induction hypothesis the label of an unsafe coercion is valid, and this blame is valid.

Induction step (property 2):

Property 2. If the label of a new expression $\texttt{<U>new C<}\overline{\texttt{T}} \Leftarrow \overline{\texttt{S}}\texttt{>(...)}$ is empty, $\texttt{C<}\overline{\texttt{T}}\texttt{>} \prec \texttt{U}$.

Case E-Coerce:

If value blame label is empty and $\texttt{C<}\overline{\texttt{T}}\texttt{>} \prec \texttt{V}$:

The result blame label is empty. By lemma 47 and lemma 13, $\texttt{C<}\overline{\texttt{W}}\texttt{>} \prec \texttt{V} \Rightarrow$ the property is satisfied.

Otherwise:

The blame label of the result value is non-empty, which satisfies the property trivially.

By property 1 of the induction hypothesis, the coercion is label is non-empty since it is an unsafe coercion.

Case E-GInvk:

$\overline{\texttt{v}}$ and $\overline{\texttt{u}}$ are passed as unmodified, and satisfy the property by induction hypothesis.

If the receiver has an empty label, by property 4, w satisfies the property (it is of form $\texttt{<C<}\overline{\texttt{T}}\texttt{>>new <}\overline{\texttt{T}} \Leftarrow \overline{\texttt{T}}\texttt{>(...)}$).

Otherwise, trivial.

Case E-DyGInvk:

Similar to E-GInvk above.

Case E-Invk:

We substitute valid values. Substitution trivially preserves the property.

Case E-FieldAcc:

This rule preserves the value $\texttt{v}_\texttt{i}$, and we can use the induction hypothesis on that.

Case E-DyFieldAcc:

Similar to E-FieldAcc.

Case E-Cast:

Similar to E-Coerce.

Case E-DynCast:

$\texttt{T} \prec \texttt{dyn}$ for all T, so this preserves the induction hypothesis.

Case E-Coerce-Fail:

Trivial.

Case E-DyGInvk-Fail:

Trivial.

Case E-DyFieldAcc-Fail:

Trivial.

Induction step (property 3):

Property 3. If the blame label of a new expression is non-empty, it has been through an unsafe coercion or an unsafe downcast, and the blame label refers to an unsafe coercion/cast.

Case E-Coerce:

There are three interesting cases.

Case value label is not empty:

The result label is equal to to the non-empty value label. By induction hypothesis, this is valid.

Case value label is empty and $\texttt{C<}\overline{\texttt{T}}\texttt{>} \prec \texttt{V}$:

The result is label is empty, but since the coercion is safe, this preserves property 3.

Case value label is empty and not $\texttt{C<}\overline{\texttt{T}}\texttt{>} \prec \texttt{V}$:

The coercion is unsafe. By property 1, the coercion label is non-empty. The result gets the coercion label, which preserves property 3.

Case E-GInvk:

Trivial (preserves the label).

Case E-DyGInvk:

Trivial (preserves the label).

Case E-Invk:

It is easy to see that this preserves all labels of values.

Case E-FieldAcc:

Trivial (preserves the label of the result).

Case E-DyFieldAcc:

Trivial (preserves the label of the result).

Case E-Cast:

Similar to E-Coerce.

Case E-DynCast:

Trivial (preserves the label).

Case E-Coerce-Fail:

Trivial.

Case E-DyGInvk-Fail:

Trivial.

Case E-DyFieldAcc-Fail:

Trivial.

Induction step (property 4):

Property 4. If label is empty in $<U>\mathtt{new}\ C<\overline{T} \Leftarrow \overline{S}>(\ldots)$, then $\overline{S} = \overline{T}$.

Case E-Coerce:

There are three interesting cases.

Case value label is not empty:

Trivial.

Case value label is empty and $C<\overline{T}> \prec V$:

The result is label is empty.

Now $meet(C<\overline{T}>, V) = C<\overline{T}>$ (by lemma 47 and lemma 13)
$\Rightarrow$ the result satisfies property 4.

Case value label is empty and not $C<\overline{T}> \prec V$:

The coercion is unsafe. By property 1, the coercion label is non-empty. The result gets the coercion label, which preserves property 4.

Case E-GInvk:

Trivial (preserves the value).

Case E-DyGInvk:

Trivial (preserves the value).

Case E-Invk:

It is easy to see that this preserves all values.
(IDEA: Use substitution lemma?)

Case E-FieldAcc:

Trivial (preserves the value).

Case E-DyFieldAcc:

Trivial (preserves the value).

Case E-Cast:

Similar to E-Coerce.

Case E-DynCast:

Trivial (preserves the value).

Case E-Coerce-Fail:

Trivial.

Case E-DyGInvk-Fail:

Trivial.

Case E-DyFieldAcc-Fail:

Trivial.

Definition 13. (safe and unsafe downcasts)

Assume $\Delta, \Gamma \vdash e : S$. The downcast $(C<\overline{T}>)e$ is safe if $map_{\downarrow}(S, C, \overline{U}) = C<\overline{T}>$ for arbitrary $\overline{U}$; otherwise, the cast is unsafe.

Definition 14. (fully typed programs)

Fully-typed programs do not containg the type dyn.

Corollary.

Fully-typed programs never depend on type similarity.

Theorem 2. All coercions are trivial when evaluating fully-typed programs, and their evaluation always terminates without blame if they have no unsafe downcasts.

Proof.

Use lemma 57 for safety of transformation (base case).

Then use induction on evaluation relation to show that these properties hold, assuming no unsafe downcasts:

1. All coercions generated during evaluation are trivial.
2. All values have form $<U>\mathtt{new}\ C<\overline{T} \Leftarrow \overline{T}>(\ldots)$ with empty label; $C<\overline{T}> <: U$ and U is precise (and thus also $C<\overline{T}>$ is precise).
3. The evaluation never terminates in blame ℓ.

We also depend on lemma 61 (trivial coercions are safe). We can use this lemma, since this theorem is a specific instance of the general blame theorem.

Use induction to show that evaluation preserves the invariants. There is always progress by lemma 60.

Base case:

It is easy to see that program after transformation satisfies the conditions.

Induction step:

Case E-Coerce:

$U <: V$ (by induction hypothesis; all generated coercions are safe and no imprecise types); use similar argument as for E-Cast.

Case E-Coerce-Fail:

Impossible, unsafe coercion. By lemma 61 and property 1 all coercions are safe.

Case E-GInvk:

$C<\overline{U}> <: D<\overline{T}>$ (by induction hypothesis) $\Rightarrow C<\overline{U}> \prec_{co} D<\overline{T}>$ (by lemma 12).

$join(C<\overline{U}>, D<\overline{T}>) \prec_{=} D<\overline{T}>$ (by lemma 40). Since both types are precise, $join(C<\overline{U}>, D<\overline{T}>) = D<\overline{T}>$.

Due to our assumptions there is no mixed inheritance (validOverride is true only if signatures are equal, and no dyn types in signatures).

All method types must thus be equal, and all coercions are safe. Signatures cannot have dyn types, and neither do receiver types, so coercions must be of form $T \Leftarrow T$ for precise type T.

Case E-Invk:

It is easy to show that substitution preserves our properties (including substitution of "this" due to induction hypothesis); this is similar to lemma 59.

Case E-FieldAcc:

The <$W_i \Leftarrow R_i$> coercion is trivial.

Since fields cannot be overridden, the coercion <$V_i \Leftarrow W_i$> is also trivial (identical types; by lemma 39 and property 2).
The receiver type is precise by induction hypothesis.

Case E-Cast:

It is easy to see that in a fully-typed program all casts are either upcasts or downcasts

If the cast is an upcast (U <: D<V>):

C<$\overline{W}$> = C<$\overline{T}$>, since C<$\overline{T}$> <: U (induction hypothesis) and U <: D<$\overline{V}$> and properties of meet (lemma 12 and lemma 47).

$\Rightarrow$ C<$\overline{W}$> <: D<$\overline{V}$> (by transitivity of <:).

else (the cast is an downcast):

We assumed above that there are no unsafe downcasts or failed casts.

Now D<$\overline{V}$> <: U. As the cast is safe, $\text{map}_\downarrow$(U, D) = D<$\overline{V}$> (by definition 13).

As the cast is successful, there exists meet(C<$\overline{T}$>, D<$\overline{V}$>) = C<$\overline{W}$>.

Now D<$\overline{V}$> <: U and C<$\overline{T}$> <: U, and C is a subclass of D.
Since $\text{map}_\downarrow$(U, D) = D<$\overline{V}$>, it is easy to see that C<$\overline{T}$> <: D<$\overline{V}$> (by definition of $\text{map}_\downarrow$ and <:)

(The last observation, C<$\overline{T}$> <: D<$\overline{V}$>, needs further elaboration. If this was not the case, there would have to be D<$\overline{R}$> <: U so that C<$\overline{T}$> <: D<$\overline{R}$> and $\overline{R} \neq \overline{V}$; but this is impossible since $\text{map}_\downarrow$(U, D) has no $\star$ components.)

By the above and lemma 47, meet(C<$\overline{T}$>, D<$\overline{V}$>) = C<$\overline{T}$>, i.e. C<$\overline{W}$> = C<$\overline{T}$>.

So the cast preserves property 2; the cast also creates no coercions, preserving property 2 and it also trivially preserves property 3.

Case E-DyGInvk:

Impossible (cannot have <dyn>new ... receiver by induction hypothesis).

Case E-DyGInvk-Fail:

Trivial.

Case E-DyFieldAcc:

Impossible (cannot have <dyn>new ... by induction hypothesis).

Case E-DyFieldAcc-Fail:

See above.

Case E-DyCast:

Impossible.

14 Properties of alternative evaluation semantics

Theorem 3.

An alternative semantics that does not track the static type of values is bisimilar to the original semantics. I.e. new expression have form new C<$\overline{T} \Leftarrow \overline{S}$>(...) instead of <U>new C<$\overline{T} \Leftarrow \overline{S}$>(...).

Proof.

It is easy to see that if we attach the runtime type of receiver type to the each guarded method invocation and field access, we can show bisimilarity.

Case E-Coerce:

Trivial.

Case E-Coerce-Fail:

Trivial.

Case E-GInvk:

Use the attached receiver type.

Case E-DyGInvk:

Use the attached receiver type.

Case E-DyGInvk-Fail:

Use the attached receiver type.

Case E-Invk:

As the runtime representation of the original receiver type arguments is identical to the static type, we can use those instead of the current static type.

Case E-FieldAcc:

Use the attached receiver type.

Case E-DyFieldAcc:

Use the attached receiver type.

Case E-DyFieldAcc-Fail:

Use the attached receiver type.

Case E-Cast:

Trivial.

Case E-DynCast:

Trivial.

Lemma 62. A trivial coercion <T ⇐ S> in the alternative semantics has no effect on evaluation.

Corollary.

These coercions can be omitted in transformation and evaluation.

Proof.

Similar to lemma 61. As the value label is preserved and the coercion is always successful, the result value in the alternative semantics will be identical to the original value.

Lemma 63. A <T ⇐ S> in the without-blame semantics has no effect on evaluation if S $\prec_{co}$ T.

Corollary.

These coercions can be omitted in transformation and evaluation.

Proof.

The proof uses the evaluation rules E-Coerce and E-Coerce-Fail, the properties of meet and the typing rule TT-Creat.

Let the target value be new C<$\overline{V}$ ⇐ $\overline{W}$>(...). By TT-Creat and TT-Coerce, C<$\overline{V}$> $\prec_{co}$ S. Now, by transitivity of $\prec_{co}$ and lemma 47, meet(C<$\overline{V}$>, T) = C<$\overline{V}$> $\Rightarrow$ the coercion evaluates to the original value.

Theorem 4.

A semantics that does not keep track of blame labels is bisimilar to the alternative semantics, modulo reported blame labels.

Proof.

Case E-Coerce:

The coercion label and value label only affect the value of the result value; they do not otherwise affect the semantics.

Case E-Coerce-Fail:

The label is only used for reporting blame; but this deviation is acceptable by our premises.

Case E-GInvk:

Labels are only used as coercion labels; they do not directly affect the evaluation step.

Case E-DyGInvk:

This is similar to E-GInvk above.

Case E-DyGInvk-Fail:

This is similar to E-Coerce-Fail above.

Case E-Invk:

The value label is simply passed on; it does not directly affect the evaluation step.

Case E-FieldAcc:

This is similar to E-GInvk above.

Case E-DyFieldAcc:

This is similar to E-GInvk above.

Case E-DyFieldAcc-Fail:

This is similar to E-Coerce-Fail above.

Case E-Cast:

This is similar to E-Coerce above.

Case E-DynCast:

Use reasoning similar to E-Invk above.